# 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.

**Tradisjonsmat**: Sortering

**Tannhelse**: Grådig søk

**Klatrevegger**: Grafer, union find

**Romfart**: Dynamisk programmering, binærsøk, interaktiv oppgave

**Logistikkdrøm**: Dynamisk programmering, segmenttrær

# Tradisjonsmat

I oppgaven får man oppgitt en liste med størrelser på bollehalvdeler og skal sette sammen like store boller.

For å løse denne oppgaven må man komme frem til at størrelsen på hver bolle nødvendigvis må være lik summen av den minste og største halvdelene: Hvis den største halvdelen parres med noen annen en den minste halvdelen, er det umulig å parre den minste halvdelen med en stor nok annen halvdel, da alle bollene skal ha lik størrelse/vekt.

Man kan dermed sortere listen og iterere gjennom å sette sammen bollehalvdeler. Hvis man får en bolle som ikke har samme størrelse, er det umulig.

```
# N, vi trenger ikke denne
N = input()

# Leser inn alle boller, legger dem sammen med id, slik at vi får en liste (0, a), (1,
b), etc
# Denne sorteres deretter på størrelse, som ligger i x[1]
h = sorted([*enumerate([*map(int,input().split())])], key=lambda x:x[1])

# w er vekten til hver bolle, dette er summen av vekta til minste og største bolle
w = h[0][1] + h[-1][1]

# Lagrer bollehalvdelpar her
p = []

# Tar hver halvdel av lista, den siste blir snudd [::-1]
for a, b in zip(h[:len(h)//2], h[len(h)//2:][::-1]):
    # Hvis halvdelene ikke summerer til riktig vekt er det umulig
    if a[1] + b[1] != w:
        print(-1)
        exit(0)

    # Gyldig kombinasjon, lagre den
    p.append((a[0], b[0]))

# Skriv ut gyldige boller
for i, j in p:
    print(i, j)
```

# Tannhelse

I denne oppgaven får man oppgitt en rekke med tenner og om de er skadet eller ikke, og skal avgjøre hvor mange av dem som kan ha vært infisert, gitt at en infisert tann fører til skade på minst én av tennenen som er tilstøtene eller tanna selv.

Man skal regne ut både minste og største antall som kan ha vært skadet.

Det største antallet er gitt ved at enhver tann som enten er skadet selv eller er nabo med en skadet tann, selv kan ha vært infisert. En tann som ikke er skadet eller har skadde naboer, kan ha vært infisert, så tellingen må være optimal.

Minste antall skadde tenner kan regnes ut ved at man iterer gjennom tennene, og markerer **neste tann** som infisert hvis man finner en skadet tann som ikke er dekket av av annen infisert tann. Vi gir her et kort bevis for at algoritmen er optimal. Ante for motsigelse skyld at algoritmen ikke er optimal. La da $i$ være indeksen til tannen vi ser på hvor algoritmen gjør den *første* feilen ved å anta at tann $i+1$ har vært infisert. Men å i stedet anta at tann $i-1$ eller $i$ har vært infisert, kan umulig gjøre at vi får en bedre løsning, da vi allerede er garantert at vi ikke har gjort et sub-optimalt trekk til venstre for tann $i$, og alle tennene til venstre for tann $i$ som er skadet er "dekket" av en infisert tann.

```
n = int(input())

teeth = [x == "1" for x in input().split()]

def count_maximum(teeth):
    # Assuming teeth padded
    assert not teeth[0] and not teeth[-1]

    # Maximum is when every tooth that is either damaged or adjacent to a damaged tooth
is infected
    count = 0

    for i in range(1, len(teeth) - 1):
        # Checks if the tooth before, the tooth after or the tooth itself is infected
        adjacent_or_infected = any(teeth[i-1:i+2])

        count += adjacent_or_infected

    return count

def count_minimum(teeth):
    lower_bound = 0
    last_infected = -10
    for i, b in enumerate(teeth):
        if b:
            if i > last_infected + 1:
                last_infected = i + 1
                lower_bound += 1
```

```
    return lower_bound

print(count_minimum(teeth), count_maximum([False, *teeth, False]))
```

# Klatrevegger

I denne oppgaven får man oppgitt en vektet graf som viser ulike punkter i en klatrevegg. Vektene representerer vanskelighetsgraden. Oppgaven er å finne minimum vekt, eller vanskelighetsgrad slik at grafen er sammenkoblet (connected).

En måte å gjøre dette på er å sortere kantene i stigende rekkefølge på vanskelighetsgrad, og å "legge til" en og en av disse kantene til det som først er en tom graf, og å avslutte prosessen når grafen er sammenhengende.

For å avgjøre om grafen er blitt sammenhengende kan vi for eksempel bruke bredde først-søk (BFS). Dette går derimot for tregt hvis vi har mange kanter og noder. For å få bedre kjøretid kan vi utnytte at det er ganske lite som egentlig endrer seg i grafen når vi legger til kun en kant: Vi har potensielt at to ulike sammenhengene komponenter slås sammen. For å holde styr på informasjon om sannhengene komponenter, kan vi bruke datastrukturen union find disjoint set (UFDS), som lar deg ha ulike sett med noder (de forskjellige komponentene), og holde styr på hvilke noder som er i hvilket sett, samt hvor store settene er. Datastrukturen tillater at man enkelt kan slå sammen settene (union).

Dersom man kommer til slutten uten å ha en eneste sammenhengende komponent er grafen ikke sammenkoblet og svaret er $-1$.

```
class UnionFind():
    def __init__(self, size):
        self.__parents = [i for i in range(size)]
        self.__size = [1] * size

    # Denne metoden finner parent til en node med id a
    def find(self, a):
        if self.__parents[a] == a:
            return a
        self.__parents[a] = self.find(self.__parents[a])
        return self.__parents[a]

    # Denne metoden slår sammen settene til nodene a og b
    # Den prøver å balansere størrelsen, slik at den parenten som er størst skal forbli
det
    def union(self, a, b):
        pa = self.find(a)
        pb = self.find(b)

        if pa == pb:
            return

        if self.__size[pa] > self.__size[pb]:
            self.__parents[pb] = pa
            self.__size[pa] += self.__size[pb]
        else:
            self.__parents[pa] = pb
```

```
            self.__size[pb] += self.__size[pa]

    # Denne metoden finner størrelsen på settet til en gitt node
    def union_size(self, a):
        return self.__size[self.find(a)]

N, K = map(int, input().split())

uf = UnionFind(N)

# Leser inn alle kanter i råformat: a, b, vanskelighetsgrad
edges = [(*map(int, input().split()),) for _ in range(K)]

# Sorterer på vanskelighetsgrad
edges.sort(key=lambda x : x[2])

for a, b, skill in edges:
    # Slår sammen a og b
    uf.union(a, b)

    # Oppgaven er ferdig hvis vi grafen er connected
    if uf.union_size(a) == N:
        print(skill)
        exit(0)

# Vi kom til endes uten å få en connected graf, da er det umulig
print(-1)
```

# Romfart

I denne oppgaven skal man finne frem til en ukjent avstand ved å sende ulike prober og få svar på om de landet eller ikke. Man skal minimere mengden drivstoff som er blir sendt. Denne oppgaven har flere deloppgaver, $N = 1$, $A = 1$, $B = 2$, $N \leq 15$, $B \leq 50$ og $N \geq 30$.

La oss starte med noen observasjoner, før vi dykker ned i de forskjellige løsningene og deloppgavene. Merk først at hvis $A = 1$ så har en sonde med $x$ enheter drivstoff $\frac{x}{B}$ sannsynlighet for å lande, hvor $1 \leq x \leq B$. Mer generelt, for alle verdier $1 \leq A$ har vi $\frac{x-A+1}{B-A+1}$ sannsynlighet for at sonden kommer frem. Si at vi er i en situasjon hvor vi har $N > 1$ og $A, B$ gitt. Hvis vi sender en sonde med $x$ enheter drivstoff, så vil en av to scenarioer spille ut:

1. Sonden kommer frem. Dette skjer med sannsynlighet $\frac{x-A+1}{B-A+1}$. I dette tilfellet vet vi at avstanden mellom jorden og Fremtid er maksimalt $x$.
2. Sonden kommer ikke frem. Dette skjer med sannsynlighet $1 - \frac{x-A+1}{B-A+1}$. I dette tilfellet vet vi at avstanden mellom jorden og Fremtid er minimalt $x + 1$.

I begge tilfellene har vi brukt $x$ enheter drivstoff, og noe *rekursivt* interressant skjer: Vi kan i første tilfelle betrakte den nye situasjonen som et nytt case av det originale problemet, hvor nå $N = N - 1$, $A = A$ og $B = x$. I det andre tilfellet har vi den nye situasjonen $N = N$, $A = X + 1$ og $B = B$. Dette betyr at vi kan evaluere minimert forventet bruk av drivstoff i en situasjon med $N$ sonder om vi allerede har regnet ut minimert forventet bruk av drivstoff i alle situasjoner hvor det er $N - 1$ sonder som skal sendes. Dette kan gjøres rekursivt uten *memoisering*, men det blir forferdelig tregt, så her er det lurt å bruke det som kalles dynamisk programmering. NIO sin referanseløsning for deloppgave 3 bruker denne løsningen, se mer under.

**Deloppgave 1:** $N = 1$, kan løses ved å sende en probe med $B$, altså maksimal mulig avstand. Det vil ikke lønne seg å prøve noe mindre. En intuisjon for dette er at et binærsøk vil ha $50\%$ sannsynlighet for å fungere, men vil koste $\frac{B+A-1}{2}$ som er $> \frac{B}{2}$ om $A > 1$. Altså vil et binærsøk koste *mer* enn halvparten av maks drivstoff, men bare fungere halvparten av tiden.

**Deloppgave 2:** $A = 1, B = 2$, kan man først sende en probe med 1, hvis den lander, fortsette med 1, ellers med 2. Vi kan direkte sjekke at denne strategien gir lavest mulig forventet drivstoff brukt. La $x \in \{1, 2\}$ være den reelle, ukjente, avstanden i en runde, og si at vi skal sende $n$ prober. Strategien som er å ikke prøve å sende med 1 vil koste $2n$ drivstoff totalt. Strategien som innebærer å først teste med en ener, vil enten koste $2n + 1$ om avstanden er 2 og $n$ om avstanden er 1. Dette skjer med lik sannsynlighet, så vi får $\frac{2n+1+n}{2} = \frac{1}{2} + 1.5n$ forventet totalt drivstoffforbruk, som er $\leq 2n$ hvor likhet skjer når $n = 1$ og etter dette får vi streng ulikhet, altså at strategien som velger å sende 1 først vil være bedre. For å fullføre argumentet må vi nå bare observere at om vi i det hele tatt vil sende en sonde med 1 drivstoff på et tidspunkt, bør vi gjøre det aller først.

**Deloppgave 3:** $N \leq 15, B \leq 50$, kan man bruke dynamisk programmering (DP). Observasjonene fra de første avsnittene over viser at vi kan direkte regne ut beste mulige valg, for eksempel ved å sette opp en tabell med optimalt neste trekk gitt et visst antall prober som

gjenstår og nedre og øvre grense. Denne tabellen konstrueres ved å for hver celle i den sjekke alle mulige trekk, og ta det som har lavest verdi av sannsynlighet for suksess × drivstoff brukt etterpå + sannsynlighet for feil × drivstoff brukt ved feil. Med lave verdier av $N$ og $B$ kan vi i god tid regne ut hele denne tabellen, som vil ha maksimal størrelse $N \times B \times B$, og hver verdi tar maksimalt $B$ steg å fylle ut, som gir total kjøretid $O(NB^3)$.

**Deloppgave 4:**, $N \geq 30$, finnes det ikke en full løsning siden fasit er den ideelle verdien, som ikke er mulig (med mindre man er synsk). Det finnes flere muligheter, slik som binærsøk som fungerer svært bra for høye verdier av $N$. Det finnes derimot måter å optimere binærsøket slik at man får enda lavere forventningsverdi og høyere score.

En måte å optimere et slikt binærsøk på er å alltid gi den muligheten til å velge $B$ i stedet for å fortsette søket. Vi har sett at dette lønner seg for eksempel når $N = 1$, men det kan lønne seg når $N > 1$ også. Tenk for eksempel $A = 999$, $B = 1000$ og $N = 10$. Her er det lite å tjene på å lære om grensen er 999 eller 1000, det er tryggest å velge 1000 alle de 10 gangene.

Vi kan i hvert steg regne ut om det vil lønne seg å velge $B$ alle de neste gangene, eller om det lønner seg å gjøre minst en til "splitt". Sagt på en annen måte kan vi regne ut minste verdien av $N$ hvor det lønner seg å gjøre nøyaktig en splitt og så velge (justert) øvre grense. Om vi har $N$ eller flere runder igjen, gjør vi nettopp en slik splitt.

Vår implementasjon av denne løsningen får 26.85/34 poeng på deloppgave fire og gjør det for øvrig ganske bra også på deloppgave 1,2 og 3, med totalt 63 poeng.

**Kommentarer:**

En annen tilnærming for å få mange poeng på deloppgave 4 er å bruke DP-ideen, men å ikke regne ut hele tabellen, og å eventuelt regne den ut offline. For eksempel kan man regne ut tabellen opp til for eksempel $N = 50$ og bare hvis $B$ er tilstrekkelig liten. Man kan også "komprimere" grensene, og regne ut tabellen opp til $B = 100$, og så tilpasse de faktiske dataen til de komprimerte verdiene.

Den beste løsningen som vi i NIO har skrevet selv bruker en blanding av modifisert binærsøk, og (modifisert) DP, som gir 94 poeng. Løsningen fra Andreas Jan van der Meulen får omtrent 0.3 poeng mer enn NIO sin beste løsning (men rundes ned til lik score som NIO sin løsning). Vi er usikker på maksimalt antall poeng det er mulig å få, da vi ikke har forsøkt å kjøre en eksakt DP på test-dataen.

Løsningen under får 93 poeng, og bruker eksakte løsninger for 1,2,3 og modifisert binærsøk i deloppgave 4.

```cpp
#include <float.h>
#include <ios>
#include <iostream>
#include <array>
```

```cpp
#include <assert.h>
#include <cmath>


constexpr int MAX_BOUNDS = 51;
constexpr int MAX_DP_N = 15;

struct dp_table_entry_t
{
    int move;
    double total_cost;
};

// DP table dimensions: number of probes left to send after this, lower bound, upper
bound
std::array<
    std::array<
        std::array<dp_table_entry_t, MAX_BOUNDS + 1>
    , MAX_BOUNDS + 1>
, MAX_DP_N> dp;

void build_dp_table()
{
    // If this is the last probe, we will always choose to send the upper bound
    for (int lower_bound = 1; lower_bound <= MAX_BOUNDS; lower_bound++)
    {
        for (int upper_bound = 1; upper_bound <= MAX_BOUNDS; upper_bound++)
        {
            dp[0][lower_bound][upper_bound].move = upper_bound;
            dp[0][lower_bound][upper_bound].total_cost = upper_bound;
        }
    }

    // Other non-base cases
    for (int probes_left = 1; probes_left < MAX_DP_N; probes_left++)
    {
        // Iterate lower bound downwards to ensure we have already computed needed
subproblems
        for (int lower_bound = MAX_BOUNDS; lower_bound > 0; lower_bound--)
        {
            for (int upper_bound = lower_bound; upper_bound <= MAX_BOUNDS; upper_bound+
+)
            {
                int optimal_move = -1;
                double optimal_cost = 1e100;

                // We may send a probe with all fuel amounts greater than that
                for (int move = lower_bound; move <= upper_bound; move++)
                {
                    double prob_failure = static_cast<double>(upper_bound - move) /
```

```
(upper_bound - lower_bound + 1);

                    double expected_cost = move;

                    // Failed, then we learn new lower bound
                    if (move != upper_bound)
                        expected_cost += prob_failure * dp[probes_left][move + 1]
[upper_bound].total_cost;
                    // Succeeded, then we learn new upper bound
                    expected_cost += (1.0 - prob_failure) * dp[probes_left - 1]
[lower_bound][move].total_cost;

                    if (expected_cost <= optimal_cost)
                    {
                        optimal_cost = expected_cost;
                        optimal_move = move;
                    }
                }

                dp[probes_left][lower_bound][upper_bound].move = optimal_move;
                dp[probes_left][lower_bound][upper_bound].total_cost = optimal_cost;
            }
        }
    }
}

void lazy_build_dp_table()
{
    static bool built = false;
    if (!built)
    {
        build_dp_table();
        built = true;
    }
}

bool send_probe(int fuel) {
    std::cout << "send " << fuel << std::endl;

    std::string response;
    std::cin >> response;

    return response == "landet";
}


void solve_subtask_1(int A, int B, int N)
{
    assert(N == 1);
```

```
    bool success = send_probe(B);

    assert(success);
}

void solve_subtask_2(int A, int B, int N)
{
    assert (A == 1 && B == 2);

    for (int landed = 0; landed < N;)
    {
        int fuel = B;

        // Only worth it to check if actual cost is 1 if we have more than two rounds
        if (N > 2 && A != B)
        {
            fuel = 1;
        }

        if (send_probe(fuel))
        {
            B = fuel;
            landed++;
        }
        else
        {
            A = fuel + 1;
        }
    }
}

void solve_subtask_3(int A, int B, int N)
{
    assert(N <= MAX_DP_N);

    for (int landed = 0; landed < N;)
    {
        int probes_left = N - landed - 1;
        int move = dp[probes_left][A][B].move;

        if (send_probe(move))
        {
            B = move;
            landed++;
        }
        else
        {
            A = move + 1;
        }
    }
```

```
}

double get_ev_for_split(double a, double b, int rounds_left, double split_at) {
    double split_cost = split_at;
    double split_prob = (split_at - a + 1.0) / (b - a + 1.0);
    return split_prob * (rounds_left - 1) * split_cost + split_cost + (1.0 -
split_prob) * rounds_left * b;
}

void solve_subtask_4(int a, int b, int n)
{
        int arrived = 0;
        while (arrived < n) {
            int fuel_exp = (a + b) >> 1;
            int rounds_left = n - arrived;
            int fuel;


      if (b*rounds_left >= get_ev_for_split(a, b, rounds_left, fuel_exp)) {
                fuel = fuel_exp;
            } else {
                fuel = b;
            }

            std::cout << "send " << fuel << std::endl;

            std::string rsp;
            std::cin >> rsp;
            if (rsp == "landet") {
                b = fuel;
                arrived++;
            } else if (rsp == "forsvunnet") {
                a = fuel + 1;
            }
        }
}



int main()
{
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);

    int R;
    std::cin >> R;

    lazy_build_dp_table();

    //std::cout << "done" << std::endl;
```

```
    for (int i = 0; i < R; i++)
    {
        int A, B, N;
        std::cin >> A >> B >> N;

        if (N == 1)
        {
            solve_subtask_1(A, B, N);
        }
        else if (A == 1 && B == 2)
        {
            solve_subtask_2(A, B, N);
        }
        else if (N <= MAX_DP_N && B < MAX_BOUNDS)
        {
            solve_subtask_3(A, B, N);
        }
        else
        {
            solve_subtask_4(A, B, N);
        }
    }
}
```

# Logistikkdrøm

For **Deloppgave 1** må man gjøre observasjonen at det bare er bånd som går til høyre og oppover som kan brukes. Ettersom kofferten skal komme fram etter minst mulig tid ville et bånd som går til venstre eller ned gjøre at det tar lengre tid å komme fram og vi ser derfor vekk fra dem i resten av oppgaven. Hvis båndet går opp eller til høyre vil det alltid være mulig å bruke det båndet.

I **Deloppgave 2** har vi to bånd. Det vil alltid være mulig å bruke et av båndene på lik måte som i **Deloppgave 1**, men vi kan potensielt forbedre det ved å bruke det andre båndet også. Hvis du har et opp-bånd i $(x_1, y_1)$ vil man kunne bruke det andre båndet hvis det ikke er til venstre og det er på en høyere y-verdi. Altså $x_2 \geq x_1$ og $y_2 \geq y_1 + 1$, og tilsvarende for høyre.

I **Deloppgave 3** har vi $K \leq 1000$ bånd. Det betyr at vi kan prøve oss på en $O(K^2)$ algoritme. En mulighet er å starte med å sortere alle båndene etter $y$ og så $x$. For hvert bånd lagrer vi største antall bånd som man kan bruke for å nå det punktet. Det kan vi finne ved å se på alle tidligere bånd, sjekke om hvert av dem kan brukes, og i så fall finne det båndet som kan nås med flest mulig tidligere bånd. Siden vi vet det bare er høyre og opp bånd som er relevante og vi sorterer, vil vi være sikker på at alle relevante bånd er beregnet før det vi ser på nå.

I **Deloppgave 4** har vi at $N \leq 1000$, altså at rutenettet ikke er for stort. Da kan vi lage en algoritme med $O(N^2)$ tidskompleksitet. Det vi gjør er at vi for hvert punkt beregner maksimalt antall bånd man kan bruke på en rute fra start til det punktet. Hvis man gjør det i stigende $x$ og $y$ retning vil man kunne gjøre det med at $D[x][y] = \max(D[x][y-1] + (\text{bånd}[x][y-1] \text{ er opp}), D[x-1][y] + (\text{bånd}[x-1][y] \text{ er høyre}))$.

I **Deloppgave 5** vil alle båndene peke oppover. Nå vil $N$ og $K$ være for stor til at det er kvadratiske løsninger som funker[1]. Spesielt har vi lyst på en løsning der vi kan vurdere flere, tidligere bånd på en gang og slippe å gå gjennom et om gangen. Dette kan gjøres med **Segmenttrær** eller **Fenwicktrær**, som er generelle datastrukturer som tillater to grunnleggende operasjoner i $O(\log N)$: summere et intervall i en liste og oppdatere listen.

Vi starter med å sortere båndene etter økende $y$-verdier og så synkende etter $x$-verdier. Vi har et 1D segmenttre som tar max av intervaller og har størrelse $N$. Hver gang vi har prosessert et bånd legger vi det inn i treet og tar maks av det som var der fra før. Når vi skal prosessere et nytt bånd tar vi maks på intervallet fra 0 til $x$-verdien og plusser en på det. Dette funker ettersom når vi prosesserer et bånd i $(x, y)$ har vi gått gjennom alle lavere $y$-verdier og dermed alle som kan brukes, og vi passer på at $x$-verdiene ikke er for store. Grunnen til at vi sorterte synkende etter $x$-verdier er fordi vi ikke ønsker å bruke andre som har samme $y$-verdi som det vi ser på. Når vi bare søker på intervallet $(0, x)$ unngår vi det. Å gjøre et sånt søk tar $O(\log N)$ tid, som totalt gir $O(K \log N)$.

[1]Det var immidlertid noen kvadratiske løsninger som klarte hele oppgaven ved å gjøre speedups. Dette var ikke meningen, men vanskelig å gjøre noe med etterpå.

**Deloppgave 6** er ikke så mye mer komplisert enn **Deloppgave 5**. Her har vi også bånd som går til høyre og vi må få dem til å funke med segmenttre løsningen fra **Deloppgave 5**. Måten vi gjør det er at vi sorterer fortsatt bånd på $y$-verdi, så på om båndet går opp eller til høyre. Hvis båndet går opp sorterer vi fortsatt synkende etter $x$-verdi, men hvis båndet går til høyre vil vi nå også kunne bruke andre bånd på samme $y$-verdi. Derfor sorterer vi dem stigende etter $x$-verdi. Før vi prosesserer båndene som går opp prosesserer vi båndene som går til høyre og så gjør vi det lag for lag.

```
#include<bits/stdc++.h>
using namespace std;
using arr = array<int, 3>;
constexpr int tree_size = 1 << 15;
int segtree[tree_size*2];

int query(int a, int b)
{
    a += tree_size, b += tree_size;
    int output = 0;
    while (a <= b)
    {
        if (a%2 == 1) output = max(output, segtree[a++]);
        if (b%2 == 0) output = max(output, segtree[b--]);
        a /= 2; b /= 2;
    }
    return output;
}

void update(int x, int val)
{
    x += tree_size;
    segtree[x] = max(segtree[x], val);
    for (x /=2; x>0; x/=2)
    {
        segtree[x] = max(segtree[2*x], segtree[2*x+1]);
    }
}

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

    vector<arr> edges;
    int x, y;
    string dir;
    for (int i = 0; i < k; i++)
    {
        cin >> x >> y >> dir;
```

```
        if (dir == "h") edges.push_back({x,0,y});
        if (dir == "o") edges.push_back({x,1,y});
    }

    sort(edges.begin(), edges.end(), [] (const auto& lhs, const auto&rhs) {
        if (lhs[2] != rhs[2]) return lhs[2] < rhs[2]; // Sorter y først
        if (lhs[1] != rhs[1]) return lhs[1] < rhs[1]; // Sorter på retning
        if (lhs[1] == 0) return lhs[0] < rhs[0];      // Sorter stigende for høyre
        return lhs[0] > rhs[0];                       // Sorter synkende for opp
    });

    for (auto [x, t, y] : edges)
    {
        if (t == 0) update(x+1, query(0, x)+1); // høyre
        else update(x, query(0, x)+1);          // opp
    }
    cout << 2 * (n-1) - query(0, tree_size-1);
}
```