Trik s konveksno ovojnico

Li Chao drevo

Jakob Žorž

Rešitev problema z napredno podatkovno strukturo
Maturitetna seminarska naloga iz informatike
Škofja Loka, 2024

GIMNAZIJA ŠKOFJA LOKA

Informatika
Gimnazija Škofja Loka
Slovenija
18. April 2024

Stvarno kazalo

1 Uvod

Matematika je zanimiva dejavnost, ki se ukvarja s tavtologijo. Marsikdo bi celo na prvi pogled rekel, da je matematika zelo trivialna, saj vse logično sledi iz aksiomov in se zdi, da je vse že znano. Vendar se pa izkaže, da je matematika zelo kompleksna in da se v njej skriva veliko zanimivih problemov. Tudi v tekmovalnem programiranju so najbolj pomembni algoritmi in razmisleki, ki so matematične narave in dokažejo pravilnost hitrejših algoritmov. Tudi v tem članku bomo obravnavali enega izmed teh algoritmov, ki reši problem, ki je na prvi pogled rešljiv le z izčrpnim preverjanjem vseh možnosti, se bo pa izkazalo, da obstaja veliko boljša rešitev, ki temelji na matematičnih zakonitostih. Mogoče bo medtem bralec občutil in spoznal pravo umetnost in lepoto matematike in morda celo pridobil kanček strahospoštovanja in zanimanja za to vedo.

2 Časovna zahtevnost

Preden se lotimo reševanja problema, si oglejmo, kaj je časovna zahtevnost in kako je uporabna:
Velik problem pride pri tem, da ni čas izvedbe vedno popolnoma enak; včasih je hitrejši, včasih pa počasnejši. Zelo je odvisno od drugih procesov in preostalih okoliščin, zato opišemo čas izvajanja bolj približno. Namesto, da konkretno povemo, koliko časa potrebuje algoritem, povemo, kako se čas izvajanja spreminja, z velikostjo vhoda. Večina časa je ta ocena dovolj dobra, saj hitra rešitev porabi desetkrat ali pa celo stokrat manj časa, kot je na voljo. Počasna rešitev pa porabi lahko potencialno tudi tisočkrat več časa, zato je dovolj le groba ocena.
Primeri:

- Algoritem, ki gre skozi seznam in izpiše vse elemente, ima časovno zahtevnost $O(n)$, kjer je n dolžina seznama.
- Algoritem, ki gre skozi seznam in izpiše vse pare elementov, ima časovno zahtevnost $O(n^2)$, kjer je n dolžina seznama.
- Algoritem, ki uredi seznam po vrsti, ima časovno zahtevnost $O(n \log n)$, kjer je n dolžina seznama. Dokaz časovne zahtevnost urejanja bomo tukaj opustili, saj ni pomemben del dokumenta.

Časovna zahtevnost ima tudi matematično definicijo:

Definicija Naj bo f funkcija, ki slika realna števila v realna števila. Časovna zahtevnost algoritma je $O(f(n))$, če obstajata pozitivni konstanti c in n_0, da velja:

$$\forall n \geq n_0 : \text{čas izvajanja algoritma} \leq c \cdot f(n)$$

Ta definicija izgleda precej zapletena, a je v resnici precej preprosta. Razmislek o njej je prepuščen bralcu.

3 Problem

Oglejmo si naslednji problem:
`https://cses.fi/problemset/task/2085`

Definicija Igraš igro, kjer imaš n različnih stopenj. Vsaka stopnja ima neko pošast, ki ima določeno moč in se lahko odločimo, ali jo premagamo ali pa preskočimo. Premagovanje pošasti nam vzame $s_i \cdot f$ časa, kjer je s_i moč pošasti in f naša spretnost (pozor: nižja kot je spretnost, manj časa potrebujemo). Če pošast preskočimo, nam to ne vzame časa. Ko premagamo pošast, se nam spretnost nastavi na f_i. Cilj igre je, da premagamo zadnjo pošast v čim krajšem času. Napiši program, ki dobi n - število stopenj, s_i - moč pošasti na i-ti stopnji in f_i - spretnost, ki jo dobimo, ko premagamo i-to pošast in izpiše najmanjši čas, ki ga potrebujemo, da premagamo zadnjo pošast.

Ena izmed možnih rešitev je, da gremo skozi vse možnosti in izberemo najboljšo. Čeprav je ta rešitev pravilna, je časovno precej neučinkovita in nam ne bo prinesla veliko točk na tekmovanju. Časovna zahtevnost te rešitve je $O(2^n)$, ker sta na vsaki stopnji dve možnosti: premagamo ali pa preskočimo pošast.
Precej enostaven način za spisat tako rešitev je rekurzija: definiramo funkcijo

```
int najmanjsi_cas(int stopnja);
```

, ki nam vrne najmanjši čas, ki ga potrebujemo, da premagamo vse pošasti od stopnje `stopnja` naprej, če smo že premagali pošast na trenutni stopnji (in mogoče še kakšne prej). Ključno je to, da na našo spretnost samo vpliva samo zadnja pošast, ki smo jo premagali.

```
// standardna knjiznica
#include<iostream>
#include<vector>
// definiramo krajsnjico za 64-bitno stevilo
typedef long long int64;
// da lahko opustimo "std::" predpono
using namespace std;


int n; // stevilo posasti
vector<int> moc_posasti; // moc vsake posasti
vector<int> spretnost_po; // nasa spretnost po uboju vsake posasti

// Ta funkcija vrne najmansi cas, da ubijemo zadnjo posast,
//ce zacnemo na neki stopnji na kateri smo ze ubili posast
int64 najmanjsi_cas(int stopnja) {
    // ce je stopnja zadnja, potem smo koncali igro in traja 0 sekund,
        da jo koncamo (d-uh)
    if(stopnja == n - 1)
       return 0;


    // Ocitno je nasa spretnost natanko: spretnost_po[stopnja],
    // ker smo po predpostavki funkcije ravnokar ubili posast na
        trenutni stopnji.
```

```
    int trenutna_spretnost = spretnost_po[stopnja];

    // zdaj imamo natanko "n - stopnja - 1" moznosti: da ubijemo neko
        naslednjo posast.
    // na sreco lahko po naslednjem uboju rekurzivno poklicemo to isto
        funkcijo,
    // saj pridemo do istega problema le z vecjo stopnjo

    // trenutni rezultat je nastavljen na 10^18, saj bomo vzeli minimum
        od vseh "n - stopnja - 1" izbir.
    int64 rezultat = 1e18;

    // poizkusamo vsako mozno naslednjo stopnjo, zacnemo pri "stopnja +
        1",
    // saj moramo iti naprej in koncamo pri vkljucno "n - 1", saj je to
        zadnja stopnja.
    for(int naslednja_stopnja = stopnja + 1; naslednja_stopnja < n;
        naslednja_stopnja++) {
        // ta spremenljivka hrani, koliko casa bi vzelo ce bi koncali
            igro tako,
        // da bi nasleden uboj bil v stopnji: naslednja_stopnja
        int64 trenutna_vrednost = 0;

        // to je cas, ki je potreben, da ubijemo tam bivajoco posast
        //                 VVVVV pretvorimo v int64, da ne prekoraci
            omejitev 32 bitnih spremenljivk
        trenutna_vrednost += (int64) trenutna_spretnost *
            moc_posasti[naslednja_stopnja];

        // to je cas, ki je potreben, da koncamo igro do konca
        trenutna_vrednost += najmanjsi_cas(naslednja_stopnja);

        // zdaj, ko imamo cas, potem nastavimo rezultat
        // na ta cas samo, ce je manjsi od rezultata
        rezultat = min(rezultat, trenutna_vrednost);

        // min(a,b) vrne manjso vrednost
    }

    // na koncu bo rezultat imel najmanjsi cas, saj smo sli skozi vse
        moznosti
    return rezultat;
}

int main(){
    // ni tako pomembno:
    // preberemo n in x
    int zacetna_spretnost;
    cin>>n>>zacetna_spretnost;
    // nastavimo velikost obeh seznamov na n
    moc_posasti.resize(n);
    spretnost_po.resize(n);
    // preberemo vrednosti in jih vnesemo v oba seznama
    for(int&i:moc_posasti)
```

```
        cin>>i;
    for(int&i:spretnost_po)
        cin>>i;

    // rezultat celega programa
    int64 rezultat = 1e18;

    // prvi uboj je lahko kjerkoli, gremo skozi vse mozne
    for(int prvi_uboj = 0; prvi_uboj < n; prvi_uboj++) {
        int64 trenutna_vrednost = 0;

        // to je cas, ki je potreben, da ubijemo tam bivajoco posast
        trenutna_vrednost += (int64) zacetna_spretnost *
            moc_posasti[prvi_uboj];

        // poklicemo funkcijo, da nam ugotovi koliko casa bo trajalo, da
            pridemo do konca
        trenutna_vrednost += najmanjsi_cas(prvi_uboj);

        rezultat = min(rezultat, trenutna_vrednost);
    }

    // rezultat bo na koncu vseboval najkrajsi cas
    // izpisi ga
    cout << rezultat << "\n";

    return 0;
}
```

4 Optimizacija z dinamičnim programiranjem

Hitro opazimo nekaj: Če privzamemo, da se globalne spremenljivke ne spreminjajo, potem je vrednost funkcije `najmanjsi_cas` za neko stopnjo vedno enaka. To pomeni, da ko enkrat končamo s tekom funkcije, lahko shranimo rezultat in pri naslednjem klicu funkcije preverimo, če smo že kdaj računali vrednost za to stopnjo. Če smo, potem lahko kar vrnemo že izračunano vrednost, sicer pa izračunamo vrednost in jo shranimo. To naredimo tako, da hranimo dve tabeli: eno za vrednosti, ki smo jih že izračunali in drugo, ki pove, ali smo že izračunali vrednost za neko stopnjo. Potem pa ugotovimo naslednje dejstvo: namesto, da bi funkcijo rekurzivno klicali, bi kar brez funkcije izpolnili tabelo z vrednostmi. Ta optimizacija sicer ne spremeni časovne zahtevnosti, vendar pa bistveno pohitri izvajanje programa, saj so klici funkcij počasnejši od navadnih operacij.

Časovna zahtevnost te rešitve je $O(n^2)$, saj moramo za vsako stopnjo iti skozi vse naslednje stopnje, kar je že veliko bolje kot prejšnjih $O(2^n)$.

```
// standardna knjiznica
#include<iostream>
#include<vector>
// definiramo krajsnjico za 64-bitno stevilo
typedef long long int64;
// da lahko opustimo "std::" predpono
```

```
using namespace std;


int n; // stevilo posasti
vector<int> moc_posasti; // moc vsake posasti
vector<int> spretnost_po; // nasa spretnost po uboju vsake posasti
vector<int64> najmanjsi_cas; // tabela namesto funkcije

int main(){
    // ni tako pomembno:
    // preberemo n in x
    int zacetna_spretnost;
    cin>>n>>zacetna_spretnost;
    // nastavimo velikost obeh seznamov na n
    moc_posasti.resize(n);
    spretnost_po.resize(n);
    // preberemo vrednosti in jih vnesemo v oba seznama
    for(int&i:moc_posasti)
        cin>>i;
    for(int&i:spretnost_po)
        cin>>i;

    // rezultat celega programa
    int64 rezultat = 1e18;

    // enaka koda kot prej, vendar namesto funkcije shranjujemo
         vrednosti v tabelo
    najmanjsi_cas.resize(n);
    najmanjsi_cas[n - 1] = 0;

    // zelo je pomembno, da ko racunamo stopnje, gremo od zadnje do prve,
    // saj pri racunanju uporabimo rezultate poznejsih stopenj
    for(int stopnja = n - 2; stopnja >= 0; stopnja--) {
        int trenutna_spretnost = spretnost_po[stopnja];

        int64 rezultat = 1e18;

        for(int naslednja_stopnja = stopnja + 1; naslednja_stopnja < n;
             naslednja_stopnja++) {
            int64 trenutna_vrednost = (int64) trenutna_spretnost *
                moc_posasti[naslednja_stopnja] +
                najmanjsi_cas[naslednja_stopnja];

            rezultat = min(rezultat, trenutna_vrednost);
        }

        najmanjsi_cas[stopnja] = rezultat;
    }

    // prvi uboj je lahko kjerkoli, gremo skozi vse mozne
    for(int prvi_uboj = 0; prvi_uboj < n; prvi_uboj++) {
        int64 trenutna_vrednost = 0;

        // to je cas, ki je potreben, da ubijemo tam bivajoco posast
```

```
        trenutna_vrednost += (int64) zacetna_spretnost *
            moc_posasti[prvi_uboj];

        // poklicemo funkcijo, da nam ugotovi koliko casa bo trajalo, da
            pridemo do konca
        trenutna_vrednost += najmanjsi_cas[prvi_uboj];

        rezultat = min(rezultat, trenutna_vrednost);
    }

    // rezultat bo na koncu vseboval najkrajsi cas
    // izpisi ga
    cout << rezultat << "\n";

    return 0;
}
```

5 Optimizacija s konveksno ovojnico

```
for(int stopnja = n - 2; stopnja >= 0; stopnja--) {
    int trenutna_spretnost = spretnost_po[stopnja];

    int64 rezultat = 1e18;

    for(int naslednja_stopnja = stopnja + 1; naslednja_stopnja < n;
         naslednja_stopnja++) {
        int64 trenutna_vrednost = (int64) trenutna_spretnost *
            moc_posasti[naslednja_stopnja] +
            najmanjsi_cas[naslednja_stopnja];

        rezultat = min(rezultat, trenutna_vrednost);
    }

    najmanjsi_cas[stopnja] = rezultat;
}
```

Če si bolj natančno pogledamo zgornjo kodo opazimo, da notranja zanka vresnici dobi minimum vseh vrednosti:

```
    trenutna_spretnost * moc_posasti[i] + najmanjsi_cas[i]
```

Skozi vse i na intervalu $[stopnja + 1, n - 1]$.
Če res razmislimo, ugotovimo, da je to enako kot iskanje najmanjšega y na neki množici premic pri nekem x. V tem primeru imamo premice, ki imajo smerni koeficient moc_posasti[i] in kostanto najmanjsi_cas[i]. Ko iteriramo v zunanji zanki, se vsako iteracijo doda nova premica: x * moc_posasti[stopnja + 1] + najmanjsi_cas[stopnja + 1]. Ostale premice ostanejo.
Zato lahko sprogramiramo podatkovno strukturo, ki nam hrani premice in nam vrne minimum za neki x.

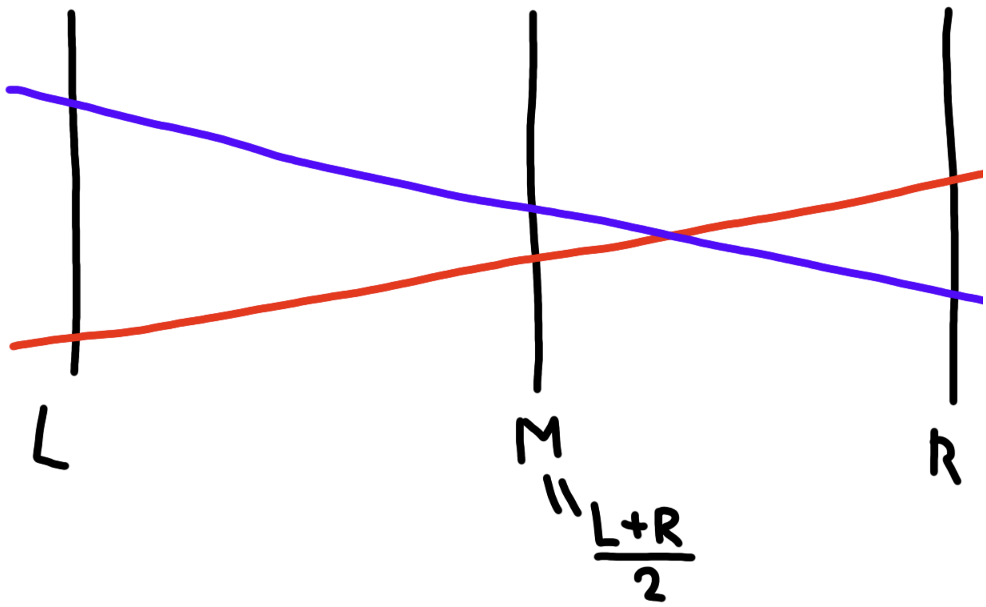
```
// standardna knjiznica
#include<iostream>
#include<vector>
// definiramo krajsnjico za 64-bitno stevilo
typedef long long int64;
// da lahko opustimo "std::" predpono
using namespace std;

// Podatkovna struktura, ki hrani premice
struct Premice {
    vector<pair<int64, int64>> premice;

    // dodaj premico y = kx + c
    void dodaj(int64 k, int64 c) {
        premice.push_back({k, c});
    }

    // pridobi nek minimum za nek x
    int64 minimum(int64 x){
        int64 rezultat = 1e18;

        // preprosto izracunaj vse y in najdi najmanjsega
        for(auto [k, c] : premice) {
            rezultat = min(rezultat, k * x + c);
        }

        return rezultat;
    }
};

int n; // stevilo posasti
vector<int> moc_posasti; // moc vsake posasti
vector<int> spretnost_po; // nasa spretnost po uboju vsake posasti
vector<int64> najmanjsi_cas; // tabela namesto funkcije

int main(){
    // ni tako pomembno:
    // preberemo n in x
    int zacetna_spretnost;
    cin>>n>>zacetna_spretnost;
    // nastavimo velikost obeh seznamov na n
    moc_posasti.resize(n);
    spretnost_po.resize(n);
    // preberemo vrednosti in jih vnesemo v oba seznama
    for(int&i:moc_posasti)
        cin>>i;
    for(int&i:spretnost_po)
        cin>>i;

    // rezultat celega programa
    int64 rezultat = 1e18;

    najmanjsi_cas.resize(n);
```

```
    najmanjsi_cas[n - 1] = 0;

    // zelo je pomembno, da ko racunamo stopnje, gremo od zadnje do prve,
    // saj pri racunanju uporabimo rezultate poznejsih stopenj

    // tokrat uporabimo podatkovno strukturo
    Premice premice;
    for(int stopnja = n - 2; stopnja >= 0; stopnja--) {
        // koda je precej kratka, saj se vecina zgodi v podatkovni
            strukturi Premice
        premice.dodaj(moc_posasti[stopnja + 1], najmanjsi_cas[stopnja +
            1]);

        najmanjsi_cas[stopnja] = premice.minimum(spretnost_po[stopnja]);
    }

    // prvi uboj je lahko kjerkoli, gremo skozi vse mozne
    for(int prvi_uboj = 0; prvi_uboj < n; prvi_uboj++) {
        int64 trenutna_vrednost = 0;

        // to je cas, ki je potreben, da ubijemo tam bivajoco posast
        trenutna_vrednost += (int64) zacetna_spretnost *
            moc_posasti[prvi_uboj];

        // poklicemo funkcijo, da nam ugotovi koliko casa bo trajalo, da
            pridemo do konca
        trenutna_vrednost += najmanjsi_cas[prvi_uboj];

        rezultat = min(rezultat, trenutna_vrednost);
    }

    // rezultat bo na koncu vseboval najkrajsi cas
    // izpisi ga
    cout << rezultat << "\n";

    return 0;
}
```

Ta koda ima še vedno časovno zahtevnost $O(n^2)$, vendar pa smo izolirali del kode, ki je odgovoren za iskanje minimuma premic. Zdaj je pa samo še naloga, da pohitrimo to podatkovno strukturo.

6 Li Chao drevo

Opazimo naslednjo stvar: če imamo neko množico premic in nek interval $[l, r]$, in želimo hitro ugotoviti minimum vseh premic za nek x na tem intervalu, lahko to naredimo z Li Chao drevesom. Najprej definiramo sredino intervala $m = \frac{l+r}{2}$ in najdemo premico, ki ima najmanjši y pri $x = m$. Rečemo ji središčna premica. Znano dejstvo je, da se dve različni premici sekata v največ eni točki, zato vemo, da če obstaja tak x, da je y neke premice manjši od y središčne premice pri istem x, potem na drugi polovici intervala, gotovo ne bo nobenega takega x. Zato lahko rekurzivno razdelimo problem na dva identična problema, vendar z intervaloma $[l, m]$ in $[m, r]$. Celotna struktura tvori binarno drevo.



Slika 1: Slika prikazuje središčno premico in neko drugo premico, ki ima manjši y pri $x = m$.

Tukaj na sliki vidimo, da je središčna premica rdeče obarvana in ima najmanjši y pri $x = m$. Vijolična premica je neka druga premica, ki po definiciji ne more imeti manjšega y pri $x = m$, zato je lahko manjša od središčne premice samo na enem intervalu.

```
// standardna knjiznica
#include<iostream>
#include<vector>
// definiramo krajsnjico za 64-bitno stevilo
typedef long long int64;
// da lahko opustimo "std::" predpono
using namespace std;

// velikost intervala iz katerega gledamo, v tem primeru vedno gledamo x
    na intervalu [1, 10^6]
const int VELIKOST = 1<<20;

typedef pair<int64,int64>Premica;

int64 vrednost_premice(int64 x, Premica l) {
    return l.first * x + l.second;
}

// Podatkovna struktura, ki hrani premice
struct Premice{
    vector<Premica> drevo;
```

```
Premice() {
    drevo = vector<Premica>(VELIKOST * 2, {0, 1e18});
}

void dodaj(int node, int l, int r, Premica premica) {
    // ce je dolzina intervala 1, ga ne moremo vec deliti
    if(l == r - 1) {
        // glede na to, da gledamo samo se na vrednosti l, lahko
            vzamemo premico, ki ima manjso vrednost in drugo zavrzemo
        if(vrednost_premice(l, premica) < vrednost_premice(l,
            drevo[node]))
            drevo[node] = premica;
        return;
    }

    // Poskrbi za robne primere
    if(premica.first > drevo[node].first)
        swap(premica, drevo[node]);

    int m = (l + r) / 2;
    // v drugem primeru pogledamo, ce je trenutna premica nasa
        premica primerna za srediscno premico
    if(vrednost_premice(m, premica) < vrednost_premice(m,
        drevo[node])) {
        // zamenjamo, da je zdaj ta srediscna in prejsnjo srediscno
            premico potisnemo dol
        swap(premica, drevo[node]);
        dodaj(2 * node, l, m, premica);
    } else
        // v nasprotnem primeru potisnemo na drugo stran
        dodaj(2 * node + 1, m, r, premica);
}

void dodaj(Premica l) {
    dodaj(1, 0, VELIKOST, l);
}

int64 minimum(int node, int l, int r, int64 x) {\
    // ce je dolzina intervala 1, ga ne moremo vec deliti
    if(l == r - 1)
        return vrednost_premice(x, drevo[node]);


    int m = (l + r) / 2;
    // najprej evaluiramo trenutno premico
    int64 val = vrednost_premice(x, drevo[node]);

    // potem pa gremo na primerno stran
    if(x < m)
        val = min(val, minimum(2 * node, l, m, x));
    else
        val = min(val, minimum(2 * node + 1, m, r, x));
    return val;
}
```

```
    int64 minimum(int64 x) {
        return minimum(1, 0, VELIKOST, x);
    }
};

int n; // stevilo posasti
vector<int> moc_posasti; // moc vsake posasti
vector<int> spretnost_po; // nasa spretnost po uboju vsake posasti
vector<int64> najmanjsi_cas; // tabela namesto funkcije

int main(){
    // ni tako pomembno:
    // preberemo n in x
    int zacetna_spretnost;
    cin>>n>>zacetna_spretnost;
    // nastavimo velikost obeh seznamov na n
    moc_posasti.resize(n);
    spretnost_po.resize(n);
    // preberemo vrednosti in jih vnesemo v oba seznama
    for(int&i:moc_posasti)
        cin>>i;
    for(int&i:spretnost_po)
        cin>>i;

    // rezultat celega programa
    int64 rezultat = 1e18;

    najmanjsi_cas.resize(n);
    najmanjsi_cas[n - 1] = 0;

    // zelo je pomembno, da ko racunamo stopnje, gremo od zadnje do prve,
    // saj pri racunanju uporabimo rezultate poznejsih stopenj

    // tokrat uporabimo podatkovno strukturo
    Premice premice;
    for(int stopnja = n - 2; stopnja >= 0; stopnja--) {
        // koda je precej kratka, saj se vecina zgodi v podatkovni
            strukturi Premice
        premice.dodaj({moc_posasti[stopnja + 1], najmanjsi_cas[stopnja +
            1]});

        najmanjsi_cas[stopnja] = premice.minimum(spretnost_po[stopnja]);
    }

    // prvi uboj je lahko kjerkoli, gremo skozi vse mozne
    for(int prvi_uboj = 0; prvi_uboj < n; prvi_uboj++) {
        int64 trenutna_vrednost = 0;

        // to je cas, ki je potreben, da ubijemo tam bivajoco posast
        trenutna_vrednost += (int64) zacetna_spretnost *
            moc_posasti[prvi_uboj];

        // poklicemo funkcijo, da nam ugotovi koliko casa bo trajalo, da
```

```
        pridemo do konca
    trenutna_vrednost += najmanjsi_cas[prvi_uboj];

    rezultat = min(rezultat, trenutna_vrednost);
  }

  // rezultat bo na koncu vseboval najkrajsi cas
  // izpisi ga
  cout << rezultat << "\n";

  return 0;
}
```

7 Zaključek

Sprehodili smo se po globinah logike in spoznali neočitno in rešili problem, ki se na prvi pogled zdi nerazrešljiv. Tako razmisleki in nove ideje so ključne v znanosti in tehnologiji, saj gonijo napredek in inovacije. Če bi se lotili kakšnega drugega problema, bi ga rešili na čisto drugačen način in bi se spet naučili nekaj novega. "Treba se je borit", pravijo. Pa naj bodo to zgolj številke na papirju ali pa pravi problemi, ki jih rešujemo. Vedno je potrebno razmišljati in se boriti za rešitve. Imeti nove ideje je po mojem mnenju ena najbolj pomembnih stvari.

8 Viri

https://cses.fi/problemset/task/2085
https://usaco.guide/plat/convex-hull-trick?lang=cpp
https://codeforces.com/blog/entry/63823
https://cp-algorithms.com/geometry/convex_hull_trick.html