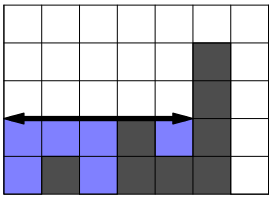
A-III-4 Kaňon

Zadanie sa síce tvári, že voda tečie spojito, je však dosť očividné, že tým sa nemusíme príliš zaťažovať. V skutočnosti totiž pri riešení zadanej úlohy vôbec nezáleží na tom, ako presne sa voda rozlieva, keď dopadne. My totiž nepotrebujeme dážď simulovať. Chceme len zistiť, ako bude vyzerať krajina v jednom konkrétnom okamihu – vo chvíli, kedy už nevie pribudnúť žiadna ďalšia voda bez toho, aby začala stúpať hladina aj v stĺpci, v ktorom prší.

Predstavme si, že v nejakom stĺpci s pršalo, až kým nenastal okamih, ktorý nás zaujíma. Čo vieme v tomto okamihu povedať o vodnej hladine? Pozrime sa na stĺpec $s+1$. Tu sú dve možnosti. Prvá možnosť: stĺpec $s+1$ je vyšší ako stĺpec s (t.j. $a_{s+1} > a_s$). V takomto prípade v tomto stĺpci zjavne nie je žiadna voda – dohora tiecť nemôže. Druhá možnosť: stĺpec $s+1$ nie je vyšší od stĺpca s. V tomto prípade musí byť v oboch rovnako vysoká vodná hladina: ani viac ani menej vody v stĺpci $s+1$ v danej chvíli zjavne nemôže byť. Ak nastala táto možnosť, môžeme teraz tú istú úvahu opakovať pre stĺpce $s+2$, $s+3$, a tak ďalej, až kým nenarazíme na nabližší stĺpec vyšší od stĺpca s. To isté platí aj v opačnom smere, teda keď sa budeme pozerať na stĺpce $s-1$, $s-2$, atď.



Dostávame teda veľmi jednoduchý popis hľadaného okamihu: ak prší v stĺpci s, hľadaný okamih nastane vtedy, keď je v okolí stĺpca s všade vodná hladina vo výške presne a_s. Táto vodná hladina navyše siaha presne od najbližšieho ostro väčšieho stĺpca vľavo od s po najbližší ostro väčší stĺpec napravo od s. Na obrázku vpravo je hrubou čiarou znázornené, odkiaľ pokiaľ bude v hľadanom okamihu siahať vodná hladina pri daždi v prostrednom stĺpci.
No a keď vieme, ako vyzerá hladina vody, ľahko zistíme množstvo vody, ktoré dovtedy naprší: v konkrétnom stĺpci i, ktorý je v zaplavenej oblasti, je presne $a_s - a_i$ vody.
Vyššie uvedené pozorovania vedú k riešeniu v čase $\Theta(n^2)$: pre každý stĺpec s pomocou cyklu nájdeme najbližší väčší stĺpec vľavo aj vpravo a popri tom si počítame množstvo vody, ktoré bude v stĺpcoch, cez ktoré prechádzame.

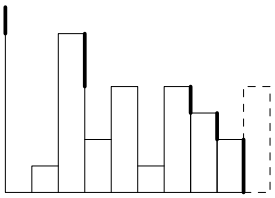
Lineárne riešenie

Aby sme predchádzajúce riešenie zrýchlili, budeme potrebovať vedieť efektívnejšie robiť dve veci: budeme potrebovať vedieť v konštantnom čase nájsť najbližší väčší stĺpec vľavo/vpravo a tak isto v konštantnom čase budeme potrebovať zistiť množstvo vody v danom intervale stĺpcov. Postupne si ukážeme obe tieto operácie.
Pozrime sa najskôr na šikovnejšie počítanie množstva vody v nejakom úseku stĺpcov. Predstavme si, že v stĺpcoch ℓ až r siaha voda do výšky v, pričom žiaden z týchto stĺpcov nemá výšku väčšiu ako v. Koľko je tam dokopy vody? V stĺpci ℓ je to $v - a_\ell$, v stĺpci $\ell+1$ je to $v - a_{\ell+1}$, a tak ďalej, dokopy je teda vody

$$(r - \ell + 1) \cdot v \;-\; (a_\ell + a_{\ell+1} + \cdots + a_r)$$

(Inými slovami, zoberieme obsah celého obdĺžnika a od neho odpočítame tú časť, ktorú tvorí zemina.)
Stačí nám teda vedieť o ľubovoľnom úseku stĺpcov povedať súčet ich výšok. Ako na to? Na začiatku si predpočítame tzv. prefixové súčty: hodnota s_i bude súčtom prvých i hodnôt postupnosti výšok stĺpcov. Všetky hodnoty s_i vieme ľahko vypočítať jedným cyklom: začneme tým, že $s_0 = 0$, a následne každé ďalšie s_{i+1} vypočítame ako $s_i + a_{i+1}$. No a akonáhle máme prefixové súčty, vieme v konštantnom čase vypočítať súčet ľubovoľného úseku pôvodnej postupnosti: zjavne totiž platí, že $a_\ell + a_{\ell+1} + \cdots + a_r$ vieme vyjadriť ako $s_r - s_{\ell-1}$.

Ostáva nám teda už len jedna otázka: potrebujeme pre každý stĺpec nájsť k nemu najbližší väčší stĺpec naľavo aj napravo. Ukážeme, ako nájsť ku každému stĺpcu najbližší väčší naľavo od neho. V programe potom tento algoritmus použijeme dvakrát: raz na pôvodnú postupnosť a_i a raz na túto postupnosť odzadu (čím nájdeme ku každému stĺpcu najbližší väčší napravo od neho).
Predstavme si, že kaňon postupne kreslíme stĺpec po stĺpci zľava doprava. Počas tohto celého procesu stojíme niekde ďaleko napravo a pozeráme sa na kaňon. Čo uvidíme? Na začiatku vidíme len ľavú stenu kaňonu. Keď nakreslíme prvý stĺpec, vidíme ten a nad ním stenu. Situáciu vo všeobecnosti si načrtneme na nasledujúcom obrázku.



Predpokladajme zatiaľ, že čiarkovane znázornený stĺpec sme ešte nenakreslili. Tejto situácii zodpovedajú tučné čiary na obrázku. Tie ukazujú, ktoré časti kaňonu vidíme, keď sa naň pozeráme sprava: celý stĺpec 9, nad ním vrch stĺpca 8, nad tým vrch stĺpca 7, ešte nad ním sú vrchné dva štvorčeky stĺpca 3, no a nad tým už je len ľavá stena kaňonu.

Čo sa teraz na tomto výhľade zmení, keď nakreslíme ďalší stĺpec? Tento stĺpec zakryje niekoľko (možno nula, možno veľa) najnižších spomedzi stĺpcov, ktoré doteraz boli viditeľné. Na obrázku sú to stĺpce 9, 8 a 7. Po nakreslení tohto stĺpca bude teda pohľad sprava obsahovať stĺpec 10, stĺpec 3 a ľavú stenu kaňonu.

Načo je nám táto informácia? To je jednoduché: priamo totiž vieme povedať, že najbližší stĺpec, ktorý je naľavo od práve pridaného stĺpca 10 a je od neho vyšší, je stĺpec 3.

Vyššie popísaný algoritmus vieme jednoducho implementovať pomocou zásobníka, v ktorom si budeme pamätať postupnosť sprava viditeľných stĺpcov kaňonu. Na začiatku do zásobníka vložíme ľavú stenu kaňonu. Postupne pre každý stĺpec s kaňonu potom opakujeme nasledovné kroky:

1. Kým je na vrchu zásobníka stĺpec s výškou $\leq a_s$, vyhodíme ho zo zásobníka (už nebude viditeľný).
2. Stĺpec, ktorý je teraz na vrchu zásobníka, si zapamätáme ako „ľavú zarážku“ pre stĺpec s.
3. Na vrch zásobníka pridáme stĺpec s.

Celý tento postup má dokopy časovú zložitosť lineárnu od n. Totiž každý stĺpec raz pridáme do zásobníka a nanajvýš raz ho z neho vyberieme.

Listing programu (C++)

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

vector<int> vyssia_vlavo (const vector<long long> &vysky) {
    stack<int> kde;          kde.push(-1);
    stack<long long> kolko;  kolko.push(1<<30);

    vector<int> odpoved;
    for (unsigned n=0; n<vysky.size(); ++n) {
        while (kolko.top() <= vysky[n]) {
            kde.pop();
            kolko.pop();
        }
        odpoved.push_back( kde.top() );
        kde.push(n);
        kolko.push(vysky[n]);
    }
    return odpoved;
}

int main() {
    int N;
    scanf("%d",&N);
    vector<long long> A(N);
    for (long long &a:A) scanf("%lld",&a);

    // predpocitame prefixove sucty
    vector<long long> P(1,0);
    for (long long a:A) P.push_back( P.back()+a );

    // predpocitame najblizsi vyssi stlpec vlavo
    vector<int> LO = vyssia_vlavo(A);

    // predpocitame najblizsi vyssi stlpec vpravo
    reverse( A.begin(), A.end() );
    vector<int> HI = vyssia_vlavo(A);
    reverse( A.begin(), A.end() );
    reverse( HI.begin(), HI.end() );
    for (int n=0; n<N; ++n) HI[n] = N-1-HI[n];

    // rovno pocitame a vypisujeme odpovede
    for (int n=0; n<N; ++n) printf("%lld%s", (HI[n]-LO[n]-1) * A[n] - P[HI[n]] + P[LO[n]+1], (n==N-1) ? "\n" : "␣");
}
```

A-III-5 Demonštrácia

Ak by v úlohe nebola podmienka o tom, že dav sa nemôže príliš zúžiť, stačilo by nám nájsť najkratšiu cestu z križovatky a do križovatky b. To by sme mohli urobiť napríklad prehľadávaním do šírky na grafe, kde vrcholy sú križovatky a hrany sú ulice, v čase $O(n + m)$. Všetky riešenia, ktoré spomenieme, sú založené na prehľadávaní do šírky, akurát vždy na nejakom upravenom grafe, ktorý nejakým spôsobom zohľadňuje podmienku o šírkach po sebe idúcich ulíc.

Prvé riešenie: graf stavov

Pri prechádzaní davu mestom sú pre nás dôležité dve veci: na ktorej križovatke sa práve nachádza a aký široký je. Usporiadanú dvojicu (v, w), kde v je číslo križovatky, na ktorej sa dav nachádza a w je šírka davu (teda šírka poslednej ulice, ktorou dav prešiel) budeme nazývať *stav*.
Zostrojíme si orientovaný graf, kde vrcholmi budú všetky možné stavy. Hrana zo stavu S_1 do stavu S_2 pôjde v našom grafe práve vtedy, keď sa dav zo stavu S_1 môže dostať do stavu S_2 prejdením jednej ulice bez toho, aby porušil pravidlo o šírke. V tomto grafe chceme nájsť dĺžku najkratšej cesty zo stavu $(a, 0)$ (v tomto stave je dav na začiatku demonštrácie: nachádza sa na križovatke a a môže vôjsť do ľubovoľnej ulice, teda má „nulovú šírku“) do ľubovoľného stavu, v ktorom sa dav nachádza na križovatke b (šírka, ktorú má dav po príchode pred sídlo veľkého vezíra nás nezaujíma). Túto dĺžku nájdeme jednoducho tak, že na našom grafe spustíme prehľadávanie do šírky zo stavu $(a, 0)$.
Ako teda zostrojíme náš graf? Nech W je šírka najširšej ulice v meste. Potom vieme, že dav bude vždy mať šírku medzi 0 a W, teda pre každú križovatku z nám stačí $W + 1$ vrcholov: $(z, 0), (z, 1), \ldots, (z, W)$. Náš graf teda bude mať $n(W + 1)$ vrcholov.
Za každú ulicu v meste potrebujeme do grafu pridať hrany, ktoré zodpovedajú prechodu davu touto ulicou. Ak je v meste ulica z križovatky x do križovatky y šírky s, potrebujeme do grafu pridať hrany z (x, w) do (y, s) pre všetky w od 0 po $s + k$. Za každú ulicu v meste teda budeme mať najviac $W + 1$ hrán v našom grafe. Počet hrán nášho grafu teda bude $O(mW)$.
Naše riešenie teda potrebuje $O((m+n)W)$ času na vytvorenie grafu. Následne strávime nanajvýš $O((m+n)W)$ času prehľadávaním, celková časová zložitosť je teda $O((m + n)W)$. Ak si naozaj budeme pamätať celý graf (aj so všetkými hranami), pamäťová zložitosť bude tiež $O((m + n)W)$.
V skutočnosti si však nepotrebujeme pamätať všetky hrany nášho grafu a stačí nám pamätať si pre každú križovatku zoznam ulíc, ktoré z nej vychádzajú. Keď potom v prehľadávaní potrebujeme zistiť, aké hrany idú z nejakého stavu (v, w), stačí nám prejsť si zoznam ulíc vychádzajúcich z križovatky v a zobrať hrany zodpovedajúce prechodom cez tie z nich, ktoré sú široké nanajvýš $w - k$ (môžete si rozmyslieť, že takéto „konštruovanie hrán za behu“ náš algoritmus nespomalí. Naopak, môže ho zrýchliť, ak pri prehľadávaní veľa vrcholov nenavštívime). Po takejto optimalizácii bude mať náš algoritmus pamäťovú zložitosť $O(nW + m)$.
Toto riešenie je teda pomerne efektívne, ak sú šírky ulíc malé a malo by zvládnuť prvých 6 sád testovacích vstupov.

Optimalizácia 1: vrcholy

Predchádzajúce riešenie je pomalé, ak sú v meste nejaké veľmi široké ulice, lebo vtedy má náš graf veľmi veľa vrcholov a jeho konštrukcia trvá dlho. V skutočnosti sa však do veľkej väčšiny z týchto stavov ani nedá dostať. Ak je totiž dav na nejakej križovatke, môže mať iba takú šírku, ako je šírka niektorej z ulíc ústiacich v tejto križovatke (prípadne nulovú, ak je to križovatka a).
Všimnime si, že stav davu je jednoznačne určený ulicou, ktorou dav prešiel ako poslednou (prípadne tým, že ešte neprešiel žiadnou ulicou). Môžeme si teda zobrať graf, kde vrcholmi sú ulice. Z ulice u nech vedie hrana do ulice v, ak mohol dav hneď po prechode ulicou u prejsť ulicou v, teda $y_u = x_v$ a $s_v \geq s_u - k$ (ulica u končí tam, kde ulica v začína a nie je oveľa širšia).
Tento graf je trochu iný ako graf, ktorý sme použili v prvom riešení, má ale jednu rovnakú dôležitú vlastnosť: sledy[1] v našom grafe korešpondujú s postupnosťami ulíc, po ktorých dav môže ísť. Preto nám stačí graf skon-

[1] *Sled* v grafe je zovšeobecnenie cesty, v ktorom sa vrcholy aj hrany môžu opakovať. Formálne je sled postupnosť, v ktorej sa striedajú vrcholy a hrany grafu, pričom na seba nadväzujú (hrana nasledujúca v slede po nejakom vrchole má začiatok v tomto vrchole a koniec vo vrchole, ktorý nasleduje po nej). Sled musí začínať aj končiť vrcholom.

štruovať a pustiť na ňom prehľadávanie do šírky, rovnako ako v prvom riešení. Za začiatok prehľadávania si môžeme buď zvoliť všetky ulice vychádzajúce z križovatky a, alebo si môžeme do grafu pridať jednu fiktívnu ulicu končiacu v a so šírkou 0 a v nej prehľadávanie začať.
Ako náš graf zostrojíme? Pre každú križovatku si vytvoríme zoznam ulíc, ktoré z nej vychádzajú. Následne vieme pre každý vrchol (ulicu) u nášho grafu nájsť všetky z neho vychádzajúce hrany tak, že prejdeme zoznam ulíc vychádzajúcich z križovatky číslo y_u a pre každú ulicu v z tohto zoznamu skontrolujeme, či $s_v \geq s_u - k$. Ak áno, potom vedie hrana z vrcholu u do v (teda po tom, čo dav prešiel ulicou číslo u mohol prejsť ulicou číslo v). Náš graf teda bude mať $O(m)$ vrcholov a $O(mD)$ hrán, kde D je maximálny počet ulíc vychádzajúcich z jednej križovatky. Časová zložitosť celého tohto riešenia je teda $O(n + mD)$ (tvorba grafu trvá $O(n + mD)$ a samotné prehľadávanie trvá $O(mD)$), pamäťová zložitosť je rovnaká. Keďže D môže byť nanajvýš m, môžeme naše zložitosti odhadnúť aj ako $O(n + m^2)$. Opäť môžeme zoptimalizovať pamäť tak, že hrany grafu budeme generovať „za behu“, v takom prípade by pamäťová zložitosť bola $O(m + n)$.
Toto riešenie by malo v časovom limite vyriešiť sady #7 a #8.

Optimalizácia 2: hrany

Naše druhé riešenie bolo pomalé, keď sa v nejakej križovatke stretávalo veľa ulíc, lebo vtedy mohol mať náš graf veľa hrán. Tento problém sa dá vyriešiť šikovným trikom.
Opäť si zostrojíme graf, kde vrcholmi budú ulice mesta. Tentoraz však v grafe budeme mať dva druhy hrán:

- Nech v je ľubovoľná ulica. V predchádzajúcom riešení sme mohli mať veľa hrán, ktoré viedli z rôznych ulíc na ulicu v. V tomto riešení budeme mať len jednu takúto hranu: spomedzi všetkých ulíc, z ktorých sme vedeli prejsť na v, vyberieme tú najširšiu ulicu u a do nášho grafu pridáme hranu **dĺžky 1** z vrcholu u do vrcholu v. (Budeme sa tváriť, že všetky ulice majú rôzne šírky. Ak sú dve ulice rovnako široké, za širšiu môžeme považovať napríklad tú s vyšším poradovým číslom.)
- Pre každú križovatku z budeme mať v grafe niekoľko hrán **dĺžky 0**, podľa nasledovného pravidla: Nech $u_1, u_2, \ldots, u_k$ sú čísla ulíc končiacich v križovatke z usporiadaných vzostupne podľa šírky. Pre každé $i \in \{1, 2, \ldots, k-1\}$ bude viesť hrana dĺžky 0 z vrcholu u_i do vrcholu u_{i+1}.

Všimnime si, že v takto zostrojenom grafe platí nasledovná skutočnosť: dav vie prejsť z ulice x na ulicu y práve vtedy, keď sa v našom grafe vieme dostať z vrcholu x do vrcholu y tak, že najskôr pôjdeme po niekoľkých (možno žiadnych) hranách dĺžky 0 a potom po hrane dĺžky 1.
Prečo je tomu tak? Nech sme naozaj vedeli prejsť z x na y. Potom v našom grafe určite existuje hrana dĺžky 1 z nejakého vrcholu x' do vrcholu y. Ulica x' je určite aspoň taká široká ako ulica x, lebo hranu dĺžky 1 vždy robíme z najširšej možnej ulice. To ale znamená, že z vrcholu x do vrcholu x' vedie postupnosť hrán dĺžky 0.
A naopak, ak sme prešli z vrcholu x najskôr nejakými hranami dĺžky 0 do vrcholu x' a až potom hranou dĺžky 1 do vrcholu y, je očividné, že v meste vieme prejsť z ulice x priamo na ulicu y.
Aký bude náš graf veľký? Počet vrcholov bude, rovnako ako v predošlom riešení $O(m)$. Do každej ulice vedie nanajvýš jedna hrana dĺžky 1 a najviac jedna hrana dĺžky 0, počet hrán teda bude tiež $O(m)$.
Na hľadanie najkratšej cesty v grafe ale tentoraz nemôžeme použiť obyčajné prehľadávanie do šírky, keďže máme hrany rôznych dĺžok. Existuje viacero možností, ako si pomôcť. V ukážkovej implementácii používame všeobecný Dijkstrov algoritmus na hľadanie najkratšej cesty v ohodnotených grafoch.[2]
Otázkou zostáva, ako náš graf skonštruujeme. Pre každú križovatku si vezmeme zoznam ulíc, ktoré v nej začínajú a zoznam ulíc, ktoré v nej končia. Oba tieto zoznamy si usporiadame podľa šírky ulíc. Následne pre každú ulicu v vychádzajúcu z križovatky nájdeme najširšiu ulicu u vchádzajúcu do križovatky takú, že po prechode cez u môže dav prejsť cez v. To môžeme urobiť napríklad binárnym vyhľadávaním.[3] Ak sa nám takú ulicu u podarí

[2] V tomto prípade existuje aj efektívnejší algoritmus, takzvané 0-1 prehľadávanie do šírky. To funguje tak, že ak ideme hranou dĺžky 0, nový vrchol zaradíme nie na koniec ale na začiatok fronty vrcholov čakajúcich na spracovanie. Časová zložitosť takéhoto prehľadávania je lineárna od veľkosti grafu. Celkovú časovú zložitosť riešenia to však nezmení, keďže už Dijkstrov algoritmus beží v čase porovnateľnom s časom potrebným na samotné zostrojenie grafu.

[3] Šikovnejším postupom sa dá z usporiadaných zoznamov ulíc zostrojiť všetky jednotkové hrany aj v lineárnom čase, no keďže už samotné triedenie potrebovalo $O(k \log k)$ času, binárnym vyhľadávaním zložitosť nepokazíme.

nájsť, pridáme do grafu hranu dĺžky 1 z u do v. Následne ešte vyššie popísaným spôsobom pridáme hrany dĺžky 0 medzi po sebe nasledujúcimi vchádzajúcimi hranami.
Spracovanie jednej križovatky a pridanie jej zodpovedajúcich hrán do grafu nám teda bude trvať čas $O(k \log k)$, kde k je počet ulíc susediacich s danou križovatkou. Keď sčítame počty ulíc susediacich s jednotlivými križovatkami, dostaneme $2m$ (lebo každú ulicu zarátame pri dvoch križovatkách), preto celá konštrukcia grafu bude trvať čas $O(n + m \log m)$. Dijkstrov algoritmus implementovaný s haldou beží v čase $O((E + V) \log E)$, kde V je počet vrcholov grafu a E je počet hrán grafu. V našom prípade teda bude bežať v čase $O(m \log m)$. Celý algroritmus teda beží v čase $O(n + m \log m)$. Pamäťová zložitosť nášho algoritmu je $O(n + m)$, nakoľko si pamätáme iba vstup, náš graf s $O(m)$ vrcholmi a $O(m)$ hranami a počas behu Dijkstrovho algoritmu ešte nejaké pomocné dátové štruktúry (pole vzdialeností a prioritnú frontu) veľkosti $O(m)$.

Listing programu uvedieme v elektronickej verzii uverejnenej na stránke súťaže.

A-III-6 Obdĺžnikový tetris

Ako prvé si pri riešení tejto úlohy musíme všimnúť, že akonáhle máme v nejakom stĺpci c zaseknutý tetrisový obdĺžnik na nejakom riadku r, všetky riadky nižšie ako r nás už nezaujímajú. Akonáhle totiž nejaký padajúci obdĺžnik bude zasahovať aj do stĺpca c, nikdy nemôže padnúť na riadok r a nižšie.
To znamená, že o každom stĺpci nám stačí vedieť jediné číslo: riadok, v ktorom je najvyššie obsadené políčko v tomto stĺpci. Keď bude nejaký obdĺžnik padať v stĺpcoch ℓ_i až $\ell_i + w_i - 1$, zasekne sa kvôli tomu stĺpcu, v ktorom je najvyššie obsadené políčko. Jeho vlastné spodné políčka budú teda o riadok vyššie. Vo všetkých týchto stĺpcoch potom narastie najvyššie obsadené políčko na rovnakú hodnotu: na riadok, v ktorom leží vrch práve pridaného obdĺžnika.
Jednoduchý program, za ktorý ste mohli získať 4 body, bude fungovať nasledovne: Pre každý padajúci obdĺžnik si prejde cez všetky stĺpce, v ktorých sa nachádza, zistí si, v ktorom z nich je najvyššie obsadené políčko. Nakoniec všetkým stĺpcom nastaví novú hodnotu najvyššieho obsadeného políčka. Časová zložitosť takéhoto riešenia bude $O(ns)$, lebo v najhoršom prípade môžu byť všetky obdĺžniky veľmi široké a my sa môžeme pozerať až na s hodnôt pre každý z nich.
Riešenia za viac bodov nebudú obsahovať žiadnu novú prevratnú myšlienku. Jediné, o čo sa budeme snažiť, je použiť vhodnú dátovú štruktúru, v ktorej budeme vedieť rýchlejšie zisťovať, aké je najvyššie číslo v po sebe idúcich stĺpcoch a nastavovať nové číslo takémuto úseku. Ďalej sa teda budeme už len rozprávať o tom, ako rýchlo zistiť najväčšie číslo z úseku poľa a ako každému prvku v úseku poľa zvýšiť hodnotu na nejaké dané číslo.

Intervalové stromy

Vhodnou štruktúrou na uchovávanie informácie o súvislých úsekoch nejakého poľa sú intervalové stromy. Pokiaľ viete, čo sú intervalové stromy a ako v nich robiť lenivé operácie, túto časť môžete preskočiť. Na opačnej strane, ak vám z tohto textu nebude jasné, ako ich implementovať, podrobnejší popis intervalových stromov môžete nájsť napríklad vo vzorovom riešení úlohy A-I-1 25. ročníka OI (časť simulácia – tretia možnosť).
Najprv si naše pole doplníme hocijakými hodnotami tak, aby malo dĺžku rovnú najbližšej väčšej mocnine dvojky. Ďalej si nad prvkami poľa postavíme úplný binárny strom. Listami tohoto stromu budú hodnoty v našom poli. V každom vnútornom vrchole tohto stromu si budeme pamätať väčšiu hodnotu z hodnôt jeho detí. Každý z vrcholov stromu bude teda obsahovať informáciu o najväčšom čísle pre nejaký súvislý úsek nášho poľa.
Ako prvé si popíšeme, ako zistiť najväčšiu hodnotu z ľubovoľného (neprázdneho) úseku S. Pozrieme sa na koreň stromu a budeme opakovať nasledovné:

1. Ak je interval vrcholu, na ktorý sa pozeráme, celý obsiahnutý v S, vrátime hodnotu tohto vrcholu.
2. Ak je interval vrcholu, na ktorý sa pozeráme, iba čiastočne obsiahnutý v S, rekurzívne sa zavoláme na obe deti tohto vrchola a vrátime väčšiu z hodnôt, ktoré nám vrátia ony.
3. Ak sa interval vrcholu, na ktorý sa pozeráme, vôbec neprekrýva s S, vrátime hodnotu, ktorá bude menšia ako akákoľvek hodnota obsiahnutá v poli (v našom prípade stačí 0).

Takýmto postupom nájdeme najväčšiu hodnotu v úseku S, pričom sa pozrieme na menej ako $4 \log_2 s$ vrcholov. Ďalej si popíšeme, ako upravovať hodnotu ľubovoľného úseku S v našom binárnom strome. Namiesto toho, aby sme upravili všetky hodnoty v poli, ktoré sa zmenia, budeme robiť lenivé úpravy v našom strome.

Lenivé úpravy

Postup bude podobný, ako pri zisťovaní hodnoty. Vo vrcholoch tretieho typu nespravíme nič, vo vrcholoch druhého typu sa rovnako rozvetvíme a nakoniec upravíme hodnotu podľa potenciálne zmenenej hodnoty detí. Vo vrcholoch prvého typu upravíme hodnotu vo vrchole a navyše si pre daný vrchol uložíme informáciu, že obom jeho deťom treba zmeniť hodnotu.
Tým sa nám aj mierne upraví zisťovanie maxima pre nejaký úsek. Vždy, keď sa pozeráme na nejaký vrchol, najprv skontrolujeme, či v ňom nemáme uloženú nevykonanú operáciu. Ak áno, najskôr ju vykonáme – čiže podľa nej upravíme obe deti a ak to nie sú listy, v každom z nich si poznamenáme, že teraz máme nevykonanú operáciu tam. Až potom budeme pokračovať podľa vyššie uvedeného postupu.
Týmto si zaručíme, že aj operácia zisťovania najväčšieho prvku v úseku, aj úprava úseku na novú hodnotu, ovplyvnia nanajvýš $O(\log s)$ vrcholov v našom strome, každý z nich v konštantnom čase. Celkový počet operácii, ktoré vykonáme, teda bude $O(n \cdot \log s)$.

Listing programu (C++)

```
#include<bits/stdc++.h>

using namespace std;

// Intervalovy strom a ulozene lenive operacie.
// Struktura:   *   (koren)                    .
//              |  \                           .
//              *   *                          .
//              | \ | \                        .
//              * * * *                        .
//              |\|\|\|\                       .
//              ********                       .
//  atd. atd.                                  .
vector <vector <int> > max_int, lazy;

// Funkcia, ktora upravi hodnotu na danom poschodi a pozicii na hodnotu v lazy
inline void lazyupdate(int floor, int position) {
    max_int [floor] [position] = lazy [floor] [position];

    // Ak nie sme na najspodnejsom poschodi, posunieme lenivu operaciu dodola
    if (floor < max_int.size() - 1) {
        lazy [floor + 1] [position * 2] = lazy [floor] [position];
        lazy [floor + 1] [position * 2 + 1] = lazy [floor] [position];
    }

    // Operaciu sme vykonali, mozeme ju vymazat
    lazy [floor] [position] = 0;
}

// Vrati najvacsi prvok na useku [begin, end)
int fetch(int floor, int position, int begin, int end) {
    // Ak treba, vykoname lenivu operaciu
    if (lazy [floor] [position]) lazyupdate(floor, position);

    // Dlzka intervalu pokryteho vrcholom a jeko [zaciatok, koniec)
    int intlen = 1<<(max_int.size() - 1 - floor), intbegin = intlen * position, intend = intbegin + intlen;
    if (intbegin >= begin && intend <= end)

        // Cely vnutri dotazovaneho intervalu
        return max_int [floor] [position];
    if (intbegin < end && intend > begin)

        // Prekryva sa s dotazovanym intervalom
        return max(
                fetch(floor + 1, position * 2, begin, end),
                fetch(floor + 1, position * 2 + 1, begin, end));
    return 0;
}

// Upravi hodnotu na useku [begin, end) na v
int update(int floor, int position, int begin, int end, int v) {
    // Tu by sa nachadzalo vykonavanie lenivej operacie. Kedze ale upravujeme
    // vzdy presne ten interval, na ktory sa predtym pytame, nie je to treba.

    int intlen = 1<<(max_int.size() - 1 - floor), intbegin = intlen * position, intend = intbegin + intlen;
    if (intbegin >= begin && intend <= end) {
```

```
        // Iba si poznacime lenivu operaciu.
        lazy [floor] [position] = v;
        return v;
    }
    if (intbegin < end && intend > begin) {

        // Update na detoch
        max_int [floor] [position] = max (
                update(floor + 1, position * 2, begin, end, v),
                update(floor + 1, position * 2 + 1, begin, end, v));
    }
    return max_int [floor] [position];
}

int main () {
    int s, n;
    scanf("%d %d", &s, &n);
    vector <int> helper(1, 0);

    // Vytvorime a naplnime vectory pre intervalac.
    for (int i = 0; (1 << i) < s; i++) {
        max_int.push_back(helper);
        lazy.push_back(helper);
        helper.resize(1<<(i+1), 0);
    }
    max_int.push_back(helper);
    lazy.push_back(helper);

    int w, h, l;
    for (int i = 0; i < n; i++) {
        scanf("%d %d %d", &w, &h, &l);

        // V tomto rieseni si pamatame najvyssie obsadene policko + 1.
        int res = fetch(0, 0, l, l+w);
        update(0, 0, l, l+w, res+h);
        printf("%d\n", res);
    }
}
```

Trikové riešenie na záver

Na záver si stručne popíšeme ešte jedno trikové riešenie, ktoré sa dá celkom príjemne implementovať len s použitím STL-kových dátových štruktúr. Počas celého behu programu si budeme udržiavať profil hracieho plánu – postupnosť neprekrývajúcich sa „plošiniek“, ktoré dohromady pokrývajú všetky stĺpce. (Plošinky = súvislé zhora viditeľné časti dna šachty a horných strán obdĺžnikov.)
Vždy, keď padne nový obdĺžnik, nájdeme si všetky plošinky, ktoré zasahujú do jeho rozsahu stĺpcov. Maximum z ich výšok nám dá výšku, v ktorej sa zasekne náš nový obdĺžnik. Všetky tieto plošinky následne zahodíme, keďže odteraz sú zakryté obdĺžnikom, ktorý práve dopadol. Výnimkou je len prvá a posledná z plošiniek – tie môžu byť novým obdĺžnikom zakryté len sčasti, takže ich namiesto úplného zahodenia len príslušne skrátime. Medzi tieto dve plošinky následne pribudne nová: horná strana práve spracovaného obdĺžnika.
(Pozor na špeciálny prípad: môže sa stať, že prvá a posledná z prekrytých plošiniek je tá istá plošinka – teda že náš obdĺžnik dopadol niekam na hornú stranu iného širšieho obdĺžnika. V takomto prípade nám z jednej starej plošinky vzniknú dve: jedna pred novou plošinkou a jedna za ňou. Za zmienku stojí, že v našom programe skracovanie plošiniek ošetrujeme tak, že toto vlastne špeciálny prípad ani nie je.)
Aby sme vedeli rýchlo zisťovať, ktoré plošinky sa kryjú s padajúcim obdĺžnikom, budeme ich mať uložené v jednom sete, usporiadané podľa súradnice začiatku. Prvú prekrývajúcu sa plošinku vieme potom získať v čase logaritmickom od počtu plošiniek, posunúť sa na ďalšiu tiež.
Tento algoritmus je v skutočnosti zhruba rovnako efektívny ako vyššie uvedené riešenie pomocou intervalového stromu. Odhad časovej zložitosti však budeme musieť spraviť šikovne.

- Plošiniek nikdy nebudeme mať viac ako n, keďže každý obdĺžnik vyrobí jednu. Plošiniek nikdy nebudeme mať viac ako s, lebo každá zaberá aspoň jeden stĺpec.
- Vždy, keď pridávame nový obdĺžnik, najviac dve plošinky skrátime a práve jednu pridáme. Toto nám pre všetky obdĺžniky dokopy zaberie $O(n \log \min(n, s))$ času.
- Vždy, keď sa nejakú inú plošinku pozrieme pri spracúvaní iného obdĺžnika, znamená to, že ju následne vyhodíme. Každú plošinku vyhodíme nanajvýš raz, preto nám aj všetko vyhadzovanie plošiniek dokopy

zaberie $O(n \log \min(n, s))$ času.

Celková časová zložitosť je teda $O(n \log \min(n, s))$.

Listing programu (C++)

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

struct segment { int left, right, height; };
bool operator< (const segment &A, const segment &B) { return A.left < B.left; }
set<segment> top_view;

int main() {
    int S, N;
    scanf("%d%d",&S,&N);
    top_view.insert( {0,S,0} );
    top_view.insert( {S,S,0} ); // zarazka
    for (int n=0; n<N; ++n) {
        int bleft, bright, bheight, bwidth;
        scanf("%d%d%d",&bwidth,&bheight,&bleft);
        bright = bleft + bwidth;

        vector<segment> covered;
        auto it = --top_view.upper_bound( {bleft,bright,0} );
        while (it->left < bright) { covered.push_back(*it); ++it; }

        int answer = 0;
        for (auto seg : covered) {
            answer = max( answer, seg.height );
            top_view.erase(seg);
        }

        if (covered.front().left < bleft) top_view.insert( { covered.front().left, bleft, covered.front().height } );
        top_view.insert( { bleft, bright, answer + bheight } );
        if (covered.back().right > bright) top_view.insert( { bright, covered.back().right, covered.back().height } );

        printf("%d\n",answer);
    }
}
```

TRIDSIATY PRVÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Eduard Batmendijn, Michal Forišek, Marián Horňák, Jaroslav Petrucha
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2016