A-I-1 Dejepis

Najskôr si stručne popíšeme jedno možné riešenie, potom si zdôvodníme, prečo funguje a nakoniec sa pozrieme na to, ako táto úloha súvisí s najkratšími cestami v grafoch.

Stručný popis riešenia

Majme už pre každú z našich t udalostí nejaký rok r_j, kedy chceme, aby sa konala.
Prechod bude operácia vyzerajúca nasledovne: Postupne prejdeme (v ľubovoľnom poradí) cez všetkých n zadaných nerovností. Každá z nich nám hovorí, že nejaká udalosť a_i musí byť aspoň d_i rokov po inej udalosti b_i. Ak momentálne táto nerovnosť nie je splnená, zvýšime r_{a_i} na presne $r_{b_i} + d_i$, čím ju splníme.
Prechod bola *úspešný*, ak sme už nič nemuseli meniť, lebo všetky nerovnosti boli splnené.
Riešenie teraz bude vyzerať nasledovne: Na začiatku nastavíme všetkých t rokov na nulu. Následne postupne spravíme t prechodov. Ak bude posledný (alebo aj hociktorý skorší) prechod úspešný, máme platné riešenie. A ak ani posledný prechod úspešný nebude, tvrdíme, že žiadne riešenie neexistuje.

Zdôvodnenie správnosti

Znázornime si zmeny, ktoré vyššie popísané riešenie spravilo, ako graf.
Presnejšie, pre každú udalosť j, ktorej rok sme niekedy zmenili, sa pozrime, kedy nastala *posledná* zmena. To muselo samozrejme byť pri kontrole nejakej nerovnosti v ktorej $a_i = j$. Zaznačme si túto zmenu tak, že si nakreslíme šípku (orientovanú hranu) z vrcholu b_i do vrcholu a_i.
Obrázok (graf), ktorý takto dostaneme, môže byť dvoch typov: buď v ňom máme nejaký cyklus alebo nie. (Cyklus je postupnosť šípiek, po ktorých sa dostaneme späť na miesto, kde sme začínali.)
Teraz tvrdíme, že platí nasledovné:

1. Ak v grafe žiaden cyklus nie je, tak posledný prechod už musel byť úspešný a teda máme platné riešenie.
2. Ak v grafe nejaký cyklus je, tak žiadne riešenie neexistuje.

Dôkaz prvého tvrdenia:
Keďže graf je acyklický a do každého vrcholu vedie nanajvýš jedna šípka, pre každý vrchol je jednoznačne určená celá cesta, ktorá doň vedie – teda postupnosť úprav, na konci ktorej tento vrchol nadobudol svoju záverečnú hodnotu. Ak napr. máme šípky $3 \to 7$, $7 \to 12$ a $12 \to 2$, tak sme niekedy počas prechodov najskôr zmenili rok udalosti 3 tak, aby bola dostatočne po udalosti 7, potom rok udalosti 12 podľa tohto nového roku udalosti 7 a niekedy ešte neskôr rok udalosti 2 podľa roku udalosti 12.
Pozrime sa na ľubovoľný vrchol j a na postupnosť šípiek $x_1 \to x_2 \to \cdots \to x_k = j$, ktoré doň vedú. Keďže udalostí (vrcholov grafu) máme t a na ceste sa vrcholy neopakujú (nemáme cykly), je $k \leq t$ a teda cestu tvorí nanajvýš $t - 1$ šípiek. Potom ale môžeme uvažovať nasledovne:

- Úpravu zodpovedajúcu prvej šípke sme určite spravili počas prvého prechodu.
- Úpravu zodpovedajúcu druhej šípke sme určite spravili najneskôr počas druhého prechodu.

  (Je možné, že nastala tiež počas prvého prechodu – ak sme nerovnosť zodpovedajúcu $x_2 \to x_3$ kontrolovali neskôr ako tú zodpovedajúcu $x_1 \to x_2$. Nemohla ale nastať neskôr: vieme totiž, že na konci prvého prechodu už mala udalosť x_2 priradený svoj finálny rok.)
- Úpravu zodpovedajúcu tretej šípke sme určite spravili najneskôr počas tretieho prechodu.
- a tak ďalej

Keďže každá cesta má nanajvýš $t - 1$ šípok, každá udalosť dostane svoj finálny rok najneskôr počas prechodu číslo $t - 1$. Počas t-teho prechodu sa teda už nič nezmení. To ale znamená, že tento prechod je úspešný a my máme platné riešenie – roky, ktoré spĺňajú všetky zadané nerovnosti.

Dôkaz druhého tvrdenia:
Vyberme si ľubovoľný jeden cyklus a označme vrcholy na ňom $x_1 \to x_2 \to \cdots \to x_k \to x_1$ tak, aby x_k bol ten naposledy zmenený. Pozrime sa teraz na šípku $x_k \to x_1$. Keďže ju máme v našom grafe, vieme, že niekedy v minulosti, keď ešte bol rok udalosti x_k menší, ako je teraz, sme museli zväčšiť rok x_1, aby bol splnený vzťah

medzi nimi. No ale keďže rok x_k sme niekedy potom zväčšili, je nutne teraz rok x_1 znovu primalý a bude ho treba pri ďalšom prechode zväčšiť. No a opakovaním tejto úvahy dostaneme, že potom bude treba zväčšiť aj rok x_2, potom aj rok x_3, a tak ďalej dokola a do nekonečna. Inými slovami, náš cyklus nám odhalil sadu k nerovníc, ktoré nevedia byť naraz všetky splnené – a teda naozaj nemôže existovať žiadne riešenie.

Grafový nadhľad

Algoritmus, ktorý sme si popísali v prvej časti tohto vzorového riešenia, má časovú zložitosť $O(tn)$: postupne robíme t prechodov, počas každého prechodu raz prejdeme cez zoznam n nerovností a každú nerovnosť spracujeme v konštantnom čase. Pamäťová zložitosť je $O(t+n)$, lebo nám stačí pamätať si vstup a aktuálny rok pre každú udalosť.
Ide o mierne upravenú verziu algoritmu Bellmana a Forda. Toho základné použitie je na hľadanie dĺžok najkratších ciest (resp. technicky presnejšie najkratších sledov[1]) z jedného vrcholu grafu do všetkých ostatných.
Tento algoritmus je síce pomalší ako známejší Dijkstrov algoritmus, má však oproti nemu jednu výhodu: funguje aj v grafoch, v ktorých majú niektoré hrany zápornú dĺžku. Ak takýto graf neobsahuje žiadny dosiahnuteľný záporný cyklus, tak najkratšie sledy sú to isté ako najkratšie cesty a Bellmanov-Fordov algoritmus ich nájde. Ak graf takýto záporný cyklus obsahuje, algoritmus to vie zdetegovať – presne ako v našom riešení, teda tým, že aj pri poslednom prechode ešte stále nastanú nejaké zmeny.

Súvis medzi našou súťažnou úlohou a najkratšími cestami v grafoch si môžeme ukázať aj názornejšie.
Namiesto nerovnosti tvaru $r_a \geq r_b + d$ uvažujme ekvivalentnú nerovnosť zapísanú nasledovne: $r_a + (-d) \geq r_b$.
Prečo sme si naše nerovnosti takto upravili? Lebo presne túto novú podobu majú nerovnosti, ktoré stretneme v úlohe o najkratších cestách v grafe. Ak najkratšia cesta zo štartu do nejakého vrcholu A má dĺžku r_a a potom z A do B vedie hrana dĺžky $(-d)$, tak použitím najkratšej cesty do A a hrany odtiaľ do B dostávame *nejakú* cestu zo štartu do B. A keďže vieme, že dĺžka tejto cesty je $r_a + (-d)$, dĺžka r_b *najkratšej* cesty do B musí byť od nej menšia alebo rovná.
Postavme si teraz nasledovný graf:

- Pre každú udalosť máme jeden vrchol.
- Pre každú nerovnosť $r_a + \ell \geq r_b$ pridáme orientovanú hranu $a \to b$ dĺžky ℓ.
- Pridáme nový „štartový“ vrchol s.
- Do každého iného vrcholu pridáme hranu $s \to a$ dĺžky 0.

(Nový pridaný vrchol je len pre naše pohodlie v ďalšom kroku. Pridané hrany z neho zodpovedajú nerovnostiam hovoriacim, že táto nová udalosť musí byť aspoň tak neskoro ako každá z pôvodných. Pridanie tejto novej udalosti teda zjavne nezmení existenciu riešenia: ľubovoľné riešenie pre pôvodné udalosti vieme doplniť tak, že novej udalosti priradíme rok rovný maximu ostatných rokov.)
Ak v tomto grafe existuje cyklus zápornej dĺžky, naša pôvodná úloha nemá riešenie – cyklus zápornej dĺžky zodpovedá sade nerovností, ktorá dokopy implikuje, že nejaký rok má byť ostro menší sám od seba.
Ak tam takýto cyklus nie je, vieme nájsť dĺžky najkratších ciest z s do všetkých ostatných vrcholov. No a ľahko nahliadneme, že ak každé r_a položíme rovné dĺžke najkratšej cesty z s do a, budú všetky nerovnosti splnené.

Listing programu (Python)

```
t, n = [ int(_) for _ in input().split() ]
A, B, D, R = [None]*n, [None]*n, [None]*n, [0]*(t+1)
for i in range(n): A[i], B[i], D[i] = [ int(_) for _ in input().split() ]

for _ in range(t):
    zmena = False
    for i in range(n):
        if R[ A[i] ] < R[ B[i] ] + D[i]:
            zmena = True
            R[ A[i] ] = R[ B[i] ] + D[i]
    if not zmena: break

if zmena: print('neexistuje')
else: print(*R[1:])
```

[1] Sled je ľubovoľná „prechádzka“ po hranách grafu. Cesta je sled, v ktorom sa neopakuje žiadny vrchol.

A-I-2 Obchodovanie

Na začiatku máme určite prázdnu loď a nulový zisk. Označme si p najväčší zisk, ktorý vieme mať s prázdnou loďou a t_i najväčší zisk, ktorý vieme mať s tým, že v lodi máme tovar i (za ktorý sme už zaplatili, ale ešte sme ho nepredali). Na začiatku teda máme $p = 0$ a všetky $t_i = -\infty$ (žiaden tovar ešte mať nevieme).
Teraz budeme postupne v chronologickom poradí spracúvať jednotlivé ponuky a udržiavať si vyššie popísané hodnoty. Pozrime sa preto presnejšie na to, ako sa tieto hodnoty zmenia, keď dostaneme ďalšiu ponuku.
Ak dostaneme ponuku na nákup tovaru i za cenu c, jediné, čo sa môže zmeniť, je hodnota t_i, teda optimálne riešenie, pri ktorom na konci máme v lodi tento tovar. Ak túto ponuku nevyužijeme, ostane t_i rovnaké ako doteraz. Ak ju využijeme, tak vieme, že tesne pred jej využitím musíme mať prázdnu loď. Aktuálna hodnota p nám hovorí, s akým najlepším celkovým ziskom sme sa vedeli dostať do tejto situácie. Nová hodnota t_i teda bude maximom zo starej hodnoty t_i (tovar i sme kúpili skôr) a hodnoty $p - c$ (mali sme prázdnu loď a zisk p a následne sme zaplatili c za tento tovar).
A ak dostaneme ponuku na predaj tovaru i za cenu c, jediné, čo sa môže zmeniť, je hodnota p. Prázdnu loď totiž vďaka tejto ponuke vieme dostať tak, že v situácii, kedy sme s optimálnym ziskom t_i mali v lodi tovar i, tento tovar teraz za cenu c predáme. Nová hodnota p je teda maximom starej hodnoty p (scenár, kedy sme už loď mali prázdnu pred touto ponukou) a hodnoty $t_i + c$ (scenár, kedy sme v lodi mali tovar i a práve sme ho predali).
Keď takto postupne prejdeme cez všetky ponuky na vstupe, hodnota p na konci bude predstavovať hľadanú odpoveď: máme prázdnu loď a najväčší možný celkový zisk.

Listing programu (Python)

```
n = int( input() )
p = 0
T = [ -10**9 for _ in range(n) ]
for _ in range(n):
    u, t, c = input().split()
    t, c = int(t)-1, int(c)
    if u == 'N':
        T[t] = max( T[t], p-c )
    else:
        p = max( p, T[t]+c )
print(p)
```

A-I-3 Domy

Označme q_i počet domov výšky i, ktoré chcú naši zákazníci. Tieto hodnoty si vieme v lineárnom čase spočítať pomocou obyčajného poľa. (Požiadavky na dom vyšší ako n zjavne stačí odignorovať.)
Pre každé $x > 1$ platia nasledovné pozorovania:

- Ak existujú domy výšky x, tak tesne pred najľavejším z nich musí byť dom výšky $x - 1$.
- Ak existujú domy výšky x, tak tesne za najpravejším z nich musí byť dom výšky $x - 1$.

Nech má najvyšší postavený dom výšku k. Ak postavíme aspoň jeden takýto dom, z našich pozorovaní vyplýva, že musíme postaviť aspoň dva domy výšky $k - 1$. A v rovnakej úvahe teraz môžeme pokračovať: pred prvým aj za posledným z domov výšky $k - 1$ musí stáť dom výšky $k - 2$. Z rovnakého dôvodu musíme potom mať aspoň dva domy výšky $k - 3$, a tak ďalej až po výšku 1.
Predstavme si teraz, že sme si zvolili k a potom pre každú výšku od 1 po k počet domov tejto výšky tak, aby boli dodržané vyššie uvedené minimálne počty a aby celkový počet domov bol n. Vieme takúto sadu domov vždy naozaj postaviť? Lahko nahliadneme, že áno: stačí napr. zobrať postupnosť $1, 2, \ldots, k, \ldots, 2, 1$ a potom do nej pre každú výšku všetky zvyšné domy tej výšky vložiť tesne za prvý dom tej výšky. Takto zjavne vždy dostaneme platné riešenie. (Postupnosť výšok domov v ňom najskôr striedavo ostáva konštantná a stúpa až po výšku k, a potom ostro klesá naspäť na výšku 1.)
Optimálne riešenie súťažnej úlohy budeme hľadať tak, že postupne vyskúšame všetky možnosti pre k (od 1 po približne $n/2$) a pre každú z nich zistíme, koľko najviac domov vieme postaviť a predať.

Pre konkrétne k túto otázku vieme pomerne priamočiaro zodpovedať v čase lineárnom od k. Najskôr postavíme všetky nutné domy (jeden výšky k a po dva každej menšej) a predáme z nich všetky, ktoré predať vieme. Nech u_k je počet zákazníkov, ktorých takto uspokojíme. Potom si spočítame z_k: počet zákazníkov, ktorí ešte stále čakajú na dom výšky nanajvýš k. Pre nich vieme postaviť ešte ďalších $n-(2k-1)$ domov a každý z nich vieme prispôsobiť ľubovoľnému z nich. Dokopy teda predáme $u_k + \min(z_k, n-(2k-1))$ domov.
Ak by sme pre každé k zvlášť počítali všetky vyššie uvedené hodnoty, dostali by sme celkovo riešenie s kvadratickou časovou zložitosťou, teda $O(n^2)$. My to však vieme spraviť aj šikovnejšie: hodnoty u_{k+1} a z_{k+1} ľahko v konštantnom čase vypočítame z hodnôt u_k a z_k. Pre každé k teda zodpovedajúce optimálne riešenie nájdeme v konštantnom čase, a keďže k nemôže byť väčšie ako zhruba $n/2$, máme celkovú časovú zložitosť $O(n)$.

Listing programu (Python)

```
n, z = [ int(_) for _ in input().split() ]
P = [ int(_) for _ in input().split() ]
Q = [0]*(n+1)
for p in P:
    if 1 <= p <= n: Q[p] += 1

najlepsie, naj_k = -1, None
uspokojeni, cakajuci = 0, 0
for k in range(1, n+1):
    # nutne postavime jeden dom vysky k a predame ho ak mozeme
    if Q[k] >= 1:
        uspokojeni += 1
    # ostatni cakajuci na vysku k uz maju sancu taky dom dostat
    cakajuci += max(0, Q[k]-1)
    # ak k > 1, nutne postavime druhy dom vysky k-1 a predame
    if k > 1 and Q[k-1] >= 2:
        uspokojeni += 1
        cakajuci -= 1
    # postavime a predame co najviac dalsich domov vysky <= k
    ostava_domov = n - (2*k-1)
    if ostava_domov < 0: break
    predame = uspokojeni + min(cakajuci, ostava_domov)
    if predame > najlepsie: najlepsie, naj_k = predame, k

# uz vieme najlepsi pocet predanych domov aj spravne k, este zostrojit spravne riesenie
# zacneme s domami ktore musime postavit
postav = [0] + [2]*(naj_k - 1) + [1]
mam = sum(postav)
# pridavame podla objednavok kym nemame n
for i in range(1, naj_k+1):
    navyse = max(0, Q[i] - postav[i])
    beriem = min(navyse, n-mam)
    postav[i], mam = postav[i]+beriem, mam+beriem
# ak stale nemame n, pridame domy vysky 1 ktore nepridame
# postavime vsetky domy okrem koncoveho klesajuceho useku
domy = [1]*(n-mam)
for i in range(1, naj_k+1): domy.extend( [i]*(postav[i]-1) )
# pridame klesajuci usek a sme hotovi
domy += list( reversed( range(1, naj_k+1) ) )
print(*domy)
```

A-I-4 Triedička

Podúloha A: minimum

Jeden možný postup, ktorým minimum nájdeme v konštantnom čase, vyzerá nasledovne:

1. Každé jadro si zistí svoje číslo a svoj vstup.
2. Usporiadame všetky jadrá podľa ich vstupu. (Minimum je teraz najviac naľavo.)
3. Jadro 0 si vypočíta kľúč 0, ostatné jadrá kľúč 1.
4. Usporiadame všetky jadrá podľa tohto kľúča.
   (Tým sa jadro 0 presunie na začiatok, poradie ostatných jadier sa nezmení.)
5. Jadro 0 sa pozrie na svoj vstup a na vstup svojho pravého suseda a vypočíta ich minimum.
   Toto minimum je riešením úlohy.

Vysvetlenie posledného kroku: Po krokoch 1 až 4 je minimum skoro vždy v pravom susedovi jadra 0. Jedinou výnimkou je situácia, kedy práve jadro 0 obsahuje minimum. V našom kroku 5 teda vždy zoberieme dve hodnoty, medzi ktorými určite je globálne minimum. Keď teda na výstup vrátime menšiu z nich, bude ňou vždy práve toto minimum.

```
číslo:          id
vstup:          input 1
                sort vstup
nula:           const 0
nenulové_id:    != číslo nula
                sort nenulové_id
vpravo:         right vstup
ja_som_minimum: < vstup vpravo
                if ja_som_minimum vstup vpravo
```

Podúloha B: najčastejší prvok

Aj túto úlohu vieme ešte riešiť na triedičke v konštantnom čase. Kľúčom k takémuto riešeniu sú nasledovné dve pozorovania: Jadro vie spoznať, že je na začiatku úseku rovnakých hodnôt – stačí sa pozrieť naľavo a overiť, že tam je iná hodnota. Takéto jadrá budeme volať aktívne. A keď už vieme, kde začínajú všetky úseky, vieme dĺžku každého úseku zistiť pomocou id aktívnych jadier. Napr. ak nejaký úsek začína jadrom s id 10 a nasledujúci úsek začína jadrom s id 17, dĺžka tohto úseku je zjavne $17 - 10 = 7$.
V programe toto môže vyzerať napr. nasledovne:

```
číslo: id
vstup: input 1

# nájdeme prvý výskyt každej hodnoty
predošlý: left vstup
aktívny: != predošlý vstup

# preusporiadame, aby boli začiatky úsekov za sebou
sort aktívny

# spočítame si dĺžky jednotlivých úsekov
za_koncom: right číslo
dĺžka: - za_koncom číslo
```

Preusporiadaním sme dostali všetky aktívne jadrá za seba (pričom ich relatívne poradie ostalo zachované). Každé z nich sa pozrie na svojho pravého suseda a z jeho a svojho čísla určí dĺžku svojho úseku.
Pozrime sa teraz na niekoľko technických detailov, ktoré treba ošetriť.

1. Netreba zabúdať, že každé jadro vykonáva všetky inštrukcie – a teda aj neaktívne jadrá si vypočítajú nejakú dĺžku. Ak si na to nedáme pozor, mohlo by nám to pokaziť zvyšok výpočtu.
2. Posledné aktívne jadro dĺžku svojho úseku vypočíta zle, lebo napravo od seba nemá jadro, ktoré by bolo „za celým poľom“. Toto potrebujeme vhodne ošetriť.
3. Pole na vstupe môže byť konštantné. V takom prípade nemáme žiadne aktívne jadrá – každé vidí cyklicky naľavo od seba tú istú hodnotu.

Ošetriť ich vieme napríklad nasledovne:

1. Hodnotu `dĺžka` prenásobíme hodnotou `aktívny`. Tým zmeníme dĺžky neaktívnych jadier na nuly a ďalej nás už nebudú trápiť.

2. Keďže aspoň jedna hodnota je na vstupe viackrát, existujú nejaké neaktívne jadrá. Posledné aktívne jadro preto vieme spoznať tak, že (cyklicky) napravo od seba má neaktívne jadro. Keď ho nájdeme, vieme jeho správnu dĺžku určiť z jeho čísla a celkového počtu jadier (ktorý je najľahšie mať v programe ako konštantu).
3. Jadro 0 prehlásime za vždy aktívne.

No a teraz už máme podobnú situáciu ako v podúlohe A: chceme nájsť najväčšiu z práve vypočítaných dĺžok a dať na výstup jej zodpovedajúci vstup. Zopakujeme teda postup, ktorý sme si vysvetlili v podúlohe A.
Kompletný program:

```
číslo:     id
vstup:     input 1
nula:      const 0
n:         const 2048
som_nula:  = číslo nula

# nájdeme prvý výskyt každej hodnoty (a ošetríme konštantné pole na vstupe)
predošlý:  left vstup
aktívny:   != predošlý vstup
aktívny:   | aktívny som_nula

# preusporiadame, aby boli začiatky úsekov za sebou
           sort aktívny

# spočítame si dĺžky jednotlivých úsekov (neaktívne jadrá dostanú nuly)
za_koncom: right číslo
dĺžka:     - za_koncom číslo
dĺžka:     * dĺžka aktívny

# ošetríme dĺžku úseku pre posledné aktívne jadro
napravo:   right aktívny
posledný:  ! napravo
posledný:  & posledný aktívny
tmp:       - n číslo
dĺžka:     if posledný tmp dĺžka

# usporiadame všetko podľa dĺžky, potom presunieme nulu na koniec
           sort dĺžka
           sort som_nula

# nula si zistí, či nemá ona maximálnu dĺžku, a podľa toho dá výstup
dĺžka_l:   left dĺžka
vstup_l:   left vstup
nula_max:  > dĺžka dĺžka_l
odpoveď:   if nula_max vstup vstup_l
```

Alternatívne riešenie podúlohy B

Keď už máme aktívne jadrá (ako vo vyššie popísanom riešení), môžeme pole preusporiadať aj ináč: najskôr spravíme `sort aktívny` a potom `sort vstup`. Prvý sort presunie všetky aktívne jadrá na koniec vstupu. Po druhom sorte máme hodnoty naspäť v poradí, v ktorom boli na začiatku, ale samotné jadrá sú v inom poradí: v každom súvislom bloku rovnakých hodnôt teraz máme najskôr všetky neaktívne jadrá (v ich pôvodnom poradí) a až nakoniec aktívne jadro.

Ak by sme napr. mali osem jadier a v nich vstupné hodnoty $(10, 20, 20, 30, 40, 40, 40, 50)$, tak aktívne jadrá budú 0, 1, 3, 4 a 7. Prvý sort dostane jadrá do poradia $(2, 5, 6, 0, 1, 3, 4, 7)$ – najskôr neaktívne a potom aktívne – a druhý do poradia $(0, 2, 1, 3, 5, 6, 4, 7)$.
Teraz vieme dĺžky úsekov vypočítať aj bez toho, aby sme museli mať explicitne v programe ako konštantu počet jadier. Každé aktívne jadro totiž môže postupovať nasledovne: ak má môj ľavý sused iný vstup ako ja, moja dĺžka je 1, a ak má rovnaký vstup, jeho číslo je posledné a moje je prvé v našom úseku, dĺžka úseku je teda ich rozdiel plus jedna.
Takto upravený program je lepší od predchádzajúceho v tom, že nepotrebuje mať ako konštantu počet jadier, na ktorom bude spustený. Máme teda jeden program, ktorý funguje pre všetky možné počty jadier naraz (až kým stačia rozsahy premenných, samozrejme).

Podúloha C: súčet

Obe predchádzajúce podúlohy sme vedeli riešiť v konštantnom čase – presnejšie, počet inštrukcií nezávisel od počtu jadier v triedičke. Teraz to už v naozaj konštantnom čase nepôjde. To preto, že nevieme rýchlo dostať naraz na to isté miesto veľa rôznych kusov informácie. V každom kroku sa totiž každé jadro vie pozrieť ku nanajvýš jednému susedovi.
Presnejšie, môžeme o výpočtoch triedičky uvažovať nasledovne. Rozdelíme si výpočet na fázy tak, že nultá fáza bude pred prvým pozretím sa k susedovi a potom v každej ďalšej fáze sa najskôr raz pozrieme ku susedovi a potom s tým, čo sme uvideli, niečo vypočítame. Po takomto rozdelení výpočtu si ľahko indukciou dokážeme, že na konci k-tej fázy každá vypočítaná hodnota závisí od vstupov v nanajvýš 2^k rôznych jadrách.
Na výpočet súčtu všetkých vstupných hodnôt teda určite treba počet fáz aspoň rovný logaritmu počtu jadier. No a to už vieme ľahko spraviť.
Pozrime sa, čo sa stane, ak sa každé jadro pozrie ku svojmu pravému susedovi a pripočíta jeho vstup ku svojmu. Ak sa teraz pozrieme na každé druhé jadro, vidíme, že jadro 0 má súčet vstupov 0+1, jadro 2 má súčet vstupov 2+3, v jadre 4 je nová hodnota súčtom vstupov 4+5, a tak ďalej. Teraz teda môžeme na jadrá s nepárnym číslom zabudnúť a ďalej pokračovať len s párnymi. Zmenšili sme tak počet jadier na polovicu a platí, že súčet ich aktuálnych hodnôt je rovný súčtu celého pôvodného vstupu.
Celé to môžeme implementovať napríklad nasledovne. Na začiatku sa každé jadro pozrie na svoj vstup a nastaví svoje číslo na svoje ID a svoj súčet na svoj vstup. Teraz postupne vykonáme niekoľko fáz, z ktorých každá bude vyzerať nasledovne:

1. Každé jadro sa pozrie ku svojmu pravému susedovi na jeho súčet.
2. Každé jadro pripočíta súčet, ktorý uvidelo, ku svojmu súčtu.
3. Každé jadro si vypočíta svoje nové číslo: párne jadrá si vydelia číslo dvoma, nepárne jadrá si zmenia číslo na 9999. (A jadrá, ktoré už mali číslo 9999 po minulej fáze, ostanú na čísle 9999.)
4. Preusporiadame jadrá podľa nových čísel. Na začiatku poľa teda dostaneme úsek polovičnej dĺžky, v ktorom sú ešte stále aktívne jadrá s číslom menším ako 9999, a za nimi sú potom všetky jadrá, ktoré už nepoužívame.

Pôvodné jadro 0 bude mať stále číslo 0, a teda stále ostane na začiatku poľa. Po jedenástich takýchto fázach sa počet aktívnych jadier zmenší z $2048 = 2^{11}$ na jediné – čiže v jadre 0 dostaneme súčet celého pôvodného vstupu.

```
# inicializácia konštánt a premenných
cislo:  id
sucet:  input 1
dva:    const 2
vela:   const 9999

# fáza, ktorú budeme 11-krát opakovať -- v plnom programe je tento blok 11x za sebou
pravy:  right sucet
sucet:  + sucet pravy
parita: % cislo dva
```

```
cislo:  / cislo dva
cislo:  if parita vela cislo
        sort cislo

# jadro 0 má správny súčet, dáme ho na výstup ako výsledok poslednej operácie
        copy sucet
```

Ďalšie alternatívne riešenie podúlohy B

V práve popísanom riešení podúlohy C vidíme, že v každej fáze si aktívne jadrá rozdelíme do dvojíc a z každej takejto dvojice vznikne jedno nové aktívne jadro. Hodnota zapamätaná v tomto aktívnom jadre je vypočítaná z hodnôt v pôvodných dvoch aktívnych jadrách.

Keby sme si celý priebeh výpočtu znázornili graficky, dostali by sme rovnaký obrázok ako pre dátovú štruktúru *intervalový strom*. Pre každé i platí, že po i fázach výpočtu každé ešte aktívne jadro zodpovedá súvislému úseku 2^i pôvodných jadier a obsahuje informáciu o celom tomto úseku.

Podobnou technikou vieme riešiť aj iné úlohy, ktoré by sme vedeli riešiť tak, že nad pôvodným poľom postavíme intervalový strom. Špeciálne takto vieme v čase logaritmickom od počtu jadier vyriešiť iným spôsobom súťažnú podúlohu B. V každom vrchole intervalového stromu (teda každom aktívnom jadre po každej fáze) si pri takomto riešení potrebujeme pamätať viacero údajov: optimálne riešenie pre jemu zodpovedajúci úsek vstupu (t.j. ktorá hodnota v ňom je najčastejšia a koľko tam má výskytov) a navyše dve čísla udávajúce, koľko rovnakých hodnôt má tento úsek na začiatku a koľko na konci.

Podotkneme ešte, že síce každej úrovni intervalového stromu zodpovedá len konštantný počet operácií, ale táto konštanta je pomerne veľká. Autorovi riešenia sa touto technikou nepodarilo zmestiť do povolených 256 inštrukcií, jeho program ich potreboval približne 350.

ŠTYRIDSIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2024