A-I-1 Po schodoch

Pri výpočte ciest hore schodami použijeme dynamické programovanie. Postupne pre každý schod si budeme počítať počet spôsobov, ktorými sa naň vieme dostať, pričom budeme využívať skôr vypočítané údaje. Tento počet pre schod i označíme p_i
Na schod 0 (teda zem pred začiatkom schodov) sa vieme dostať jedným spôsobom – tak, že nič nespravíme. Teda máme $p_0 = 1$.
Majme teraz nejaký vyšší schod i. Všetky možné spôsoby, ako sa dostať na schod i, môžeme rozdeliť do niekoľko disjunktných skupín podľa toho, z ktorého schodu j sme spravili krok na schod i. No a zo zeme na konkrétny schod j sa vieme dostať presne p_j spôsobmi. Platí teda, že p_i je rovné súčtu hodnôt p_j pre tie schody j, z ktorých vieme spraviť krok na schod i – teda cez schody, pre ktoré platí $j < i$ a $(h_{j+1} + \cdots + h_i) \leq d$.
Dobrá implementácia tohto riešenia má časovú zložitosť v najhoršom prípade kvadratickú od počtu schodov. Za toto riešenie ste mohli získať 6 bodov, prípadne dokonca 8, ak ste hľadanie vyhovujúcich j začali od $j = i - 1$ smerom dodola a prerušili ste ho akonáhle už bol priveľký rozdiel výšok i-teho a j-teho schodu.

Vzorové riešenie bude mať časovú zložitosť lineárnu od počtu schodov, teda $O(n)$. Zlepšíme predchádzajúce riešenie pomocou jednoduchého nástroja: prefixových súčtov.
Ak máme nejakú postupnosť $a_0, \ldots, a_{n-1}$, môžeme v lineárnom čase vypočítať novú postupnosť $b_0, \ldots, b_n$ definovanú nasledovne: $b_0 = 0$ a $\forall i > 0 : b_i = b_{i-1} + a_{i-1}$. Potom platí, že každé b_i je súčet prvých i členov postupnosti a.
Prefixové súčty prvýkrát použijeme na zadanú postupnosť výšok schodov h_i a vyrobíme si tak novú postupnosť v_i: „nadmorskú výšku“ jednotlivých schodov. Podmienku „viem spraviť krok zo schodu j na schod i“ teraz vieme overiť v konštantnom čase: stačí sa pozrieť, či $v_i - v_j \leq d$.

Druhým krokom nášho riešenia bude, že si pre každý schod i nájdeme najnižší schod m_i, z ktorého ešte vieme spraviť krok na schod i. Toto vieme tiež celé spraviť v lineárnom čase, a to vďaka pozorovaniu, že hodnoty m_i sú neklesajúce. Napr. ak zo schodu 2 neviem spraviť krok na schod 7, nebudem ho vedieť spraviť ani na vyšší schod 8.
Na začiatku schodov vieme, že $m_1 = 0$: zo zeme vieme spraviť krok na prvý schod. Postavme teraz Adama na schod 1 a Betku na schod 0 a dokola opakujme nasledovný proces:

- Adam sa posunie o jeden schod dohora. Toto je nový schod i.
- Kým Betka stojí na schode, z ktorého nevie spraviť krok k Adamovi, posunie sa o jeden schod dohora. Takto Betka postupne vyjde zo schodu m_{i-1} na schod m_i. Hodnotu m_i si zapamätáme.

Celý tento proces vieme odsimulovať v lineárnom čase a postupne tak zistiť všetky hodnoty m_i. (Ak vám nie je zjavné, prečo má dokopy celá simulácia tohto procesu lineárnu zložitosť, všimnite si, že Adam dokopy spraví presne $n - 1$ a Betka dokopy spraví nanajvýš $n - 1$ krokov dohora.)

Posledná úprava, ktorú spravíme v kvadratickom riešení, bude, že priebežne si spolu s hodnotami p_i budeme počítať aj ich prefixové súčty s_i.
Výpočet konkrétnej hodnoty p_i teraz bude vyzerať nasledovne: Nech $j = m_i$. Vieme, že $p_i = p_j + \cdots + p_{i-1}$. Pomocou už vypočítaných prefixových súčtov postupnosti p vieme tento súčet vypočítať v konštantnom čase: $p_i = s_i - s_j$. No a následne už len dopočítame, že $s_{i+1} = s_i + p_i$ a môžeme ísť na nasledujúci schod.

Na záver už len podotkneme, že hodnoty p_i môžu až exponenciálne rásť. Aby nebolo potrebné implementovať aritmetiku veľkých čísel, pridali sme do zadania vetu, že nás z čísla p_n zaujíma len zvyšok po delení $10^9 + 7$. Tento zvyšok zjavne vieme zistiť tak, že všetky medzivýsledky budeme počítať modulo $10^9 + 7$.

Listing programu (C++)

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

const int MOD = 1000000007;
```

```
int uprav(int x) {
    if (x < 0) x += MOD;
    if (x >= MOD) x -= MOD;
    return x;
}

int main() {
    int N, D;
    cin >> N >> D;
    vector<int> H(N);
    for (int &h : H) cin >> h;

    // vypocitame vysky v ktorych su jednotlive schody
    vector<long long> V = {0};
    for (int h : H) V.push_back( V.back()+h );

    // pre kazdy schod vypocitame prvy z ktoreho nan mozeme ist
    vector<int> M = {0,0};
    for (int a=2, b=0; a<=N; ++a) {
        while (V[a] - V[b] > D) ++b;
        M.push_back(b);
    }

    // postupne pocitame odpovede a aj ich prefixove sucty
    vector<int> P = {1}, S = {0,1};
    for (int i=1; i<=N; ++i) {
        P.push_back( uprav( S[i] - S[M[i]] ) );
        S.push_back( uprav( S.back() + P.back() ) );
    }
    cout << P[N] << endl;
}
```

A-I-2 Nová bankovka

Čiastočné body za túto úlohu sa dali získať rôznymi spôsobmi. Prvú sadu sa dalo vyriešiť hrubou silou – vyskúšaním všetkých možných spôsobov platenia. Druhú sadu sa dalo vyriešiť pažravým algoritmom, ktorý popisujeme nižšie. Prvé tri sady sa dalo vyriešiť algoritmom, ktorý pre každú sumu od 1 po max t_i zistí optimálny spôsob platenia pomocou dynamického programovania.

Dokážeme si najskôr jedno tvrdenie o pôvodnej sade platidiel. (Toto tvrdenie teda priamo platí aj pre eurá a analogický dôkaz zafunguje aj napr. pre české koruny alebo americké doláre. Samotné tvrdenie je intuitívne – naše sady platidiel boli navrhované tak, aby sa s nimi pohodlne pracovalo – ale jeho presný dôkaz si bude vyžadovať dôslednú argumentáciu.)
Keby sme nemali nové platidlo (a aj keby sme ho mali a malo hodnotu 100 000), mohli by Kocúrkovčania používať jednoduchý pažravý algoritmus: Ak chceš nejakú sumu zaplatiť najmenším počtom platidiel, stačí vždy použiť najväčšie platidlo, ktorého hodnota je nanajvýš rovná sume, ktorú ešte ostáva zaplatiť.
Dôkaz spravíme od konca. Mince s hodnotou 1, 2 a 5 nazvime *malé mince*. Ak platíme sumu od 1 do 9 toliarov, môžeme použiť len malé mince. Ľahko overíme, že pre každú z týchto súm nájde vyššie popísaný pažravý algoritmus optimálne riešenie.
A naopak, nech platíme akúkoľvek sumu, ak sme použili optimálny počet platidiel, tak malé mince, ktoré sme použili, musia mať v súčte hodnotu menšiu ako 10. Prečo? Lebo v opačnom prípade (rozmyslite si detaily) by sme vždy vedeli spomedzi použitých malých mincí vybrať podmnožinu s hodnotou presne 10 alebo 11, no a namiesto týchto mincí by bolo lepšie použiť jedinú mincu s hodnotou 10, resp. ich zaplatiť ako 10+1.
Tieto dve pozorovania dokopy nám teda hovoria, že nech optimálne platíme akúkoľvek sumu, tak malými mincami zaplatíme presne jej poslednú cifru. A vieme, že to pažravý algoritmus spraví vždy optimálne. Teraz teda môžeme zabudnúť na existenciu malých mincí, vynulovať poslednú cifru sumy, ktorú platíme, a celú túto úvahu zopakovať pre ďalší rád.
Takto postupne dostaneme, že pažravý algoritmus funguje aj pre desiatky, stovky, tisíce a desaťtisíce toliarov. Dokončenie dôkazu pre rády od stotisíc vyššie už prenechávame na čitateľa.

Čo sa teraz zmení v prítomnosti nového platidla s nominálnou hodnotou x?
Ak by sme vedeli, koľkokrát pri platení sumy t použiť bankovku x, bolo by to ľahké – zvyšok sumy už totiž platíme pôvodnou sadou platidiel, a teda môžeme použiť jednoduchý pažravý algoritmus.

My to síce nevieme, ale je tu ľahká pomoc: vyskúšame „všetky“ možnosti a vyberieme najlepšiu z nich. Samozrejme, pri tom skúšaní možností sa oplatí ešte trochu porozmýšľať: napr. pre $x = 3$ si nemôžeme zrovna dovoliť skúšať vyše tristo miliónov možností.
Pomôže nám jedno z nasledovných tvrdení:
Tvrdenie 1. Pre zadané obmedzenia platí, že bankovku s hodnotou x vždy použijeme menej ako 50 000-krát.
Dôkaz. Ak $x > 50\,000$, tak ju použijeme menej ako 50 000-krát jednoducho preto, že $50\,000^2 > 2 \cdot 10^9 \geq \max t_i$. (Slovne: zaplatili by sme určite viac ako požadovanú sumu.) No a ak $x < 50\,000$, tak sa nikdy neoplatí použiť 50 000 kusov bankovky x, lebo namiesto nich je lepšie použiť x kusov bankovky s hodnotou 50 000.
Tvrdenie 2. Pre zadané obmedzenia platí, že bankovku s hodnotou x vždy použijeme menej ako 40 014-krát.
Dôkaz. Sumu $\leq 2 \cdot 10^9$ vieme bez použitia bankovky x zaplatiť na menej ako 40 014 kusov platidiel. Ak by sme bankovku x použili viackrát, mali by sme už priveľa kusov platidiel, a teda by nešlo o optimálne riešenie.

A to už je všetko. Implementácia riešenia je jednoduchá: vyskúšame všetky zmysluplné možnosti pre počet použitých bankoviek s hodnotou x a pre každú z nich spustíme pažravý algoritmus na zaplatenie zvyšku sumy zvyšnými typmi platidiel. Spomedzi všetkých takto získaných riešení si na záver samozrejme vyberieme to najlepšie.

Listing programu (Python)

```
def pazravo(t):
    platidla = [1,2,5,10,20,50,100,200,500,1000,2000,5000,10000,20000,50000]
    pocet_kusov = 0
    for p in reversed(platidla):
        pocet_kusov += t // p
        t = t % p
    return pocet_kusov

x = int( input() )
q = int( input() )
T = [ int(_) for _ in input().split() ]

for t in T:
    najlepsie = pazravo(t)
    for kolko_x in range(0,najlepsie):
        if kolko_x * x > t:
            break
        toto = kolko_x + pazravo(t - kolko_x * x)
        najlepsie = min( najlepsie, toto )
    print( najlepsie )
```

A-I-3 Rekonštrukcia mapy

V riešení budeme používať terminológiu z teórie grafov: kráľovstvo je strom, ostrovy a mosty sú jeho vrcholy a hrany, a počet mostov vedúcich z ostrova voláme stupeň vrcholu.
Pre $n = 1$ máme jeden vrchol a ten musí mať stupeň 0. Odteraz ďalej budeme predpokladať, že $n > 1$.
Keďže máme v strome $n-1$ hrán, musí platiť, že každý vrchol má stupeň nanajvýš $n-1$. Strom je súvislý, takže každý vrchol musí mať stupeň aspoň 1. Navyše vieme, že každá hrana prispieva ku stupňom dvoch vrcholov, a teda súčet stupňov všetkých vrcholov musí byť presne $2(n-1)$. Akonáhle niektorá z týchto podmienok nie je splnená, riešenie neexistuje.

Nič zložitejšie nás už nečaká – vyššie uvedená sada podmienok je totiž nie len nutná ale zároveň aj postačujúca. Inými slovami, nižšie si ukážeme, že ak sú všetky uvedené podmienky splnené, tak nejaký strom so zadanými stupňami vrcholov existuje.
Tvrdenie dokážeme matematickou indukciou od počtu vrcholov stromu.
Pre $n = 2$ existuje jediná postupnosť stupňov spĺňajúca všetky podmienky: $(1, 1)$. A pre ňu skutočne existuje aj strom: dva vrcholy spojené hranou.
Majme teraz postupnosť dĺžky n. Zjavne musí existovať nejaké i také, že $d_i = 1$ (keby všetky boli aspoň 2, bol by súčet priveľký). Nech j je index zodpovedajúci najväčšej hodnote d_j. Vyrobme novú postupnosť d' z d tak, že vynecháme d_i a o jedno zmenšíme d_j. Nová postupnosť d' spĺňa všetky podmienky (rozmyslite si prečo)

a keďže je kratšia, podľa indukčného predpokladu jej naozaj zodpovedá aj nejaký strom. Strom pre pôvodnú postupnosť teraz zostrojíme tak, že pridáme nový list a pripojíme ho hranou k vrcholu zodpovedajúcemu pôvodnej hodnote d_j.

Vyššie uvedený dôkaz nám zároveň dáva algoritmus, ako hľadaný strom zostrojiť.
Ak chceme dosiahnuť optimálnu časovú zložitosť (lineárnu od počtu vrcholov stromu), potrebujeme si ešte rozmyslieť, ako šikovne hľadať indexy i a j v každej iterácii. V našom riešení to budeme robiť tak, že si vrcholy roztriedime na $n-1$ kôpok podľa stupňa. Za vrchol i vždy vyberieme ľubovoľný so stupňom 1 a za vrchol j ľubovoľný s aktuálne maximálnym stupňom. Maximálny stupeň sa môže počas behu algoritmu len zmenšovať, a zakaždým sa zmenší nanajvýš o 1, takže každá iterácia nášho algoritmu prebehne v konštantnom čase.

Listing programu (Python)

```
import sys

N = int( input() )
D = [ int(_) for _ in input().split() ]

if min(D) < 1 or max(D) > N-1 or sum(D) != 2*N-2:
    print('neexistuje')
    sys.exit()

stupen_na_vrcholy = [ [] for _ in range(N) ]
for n in range(N):
    stupen_na_vrcholy[ D[n] ].append(n)

max_stupen = max(D)

for kolo in range(N-1):
    i = stupen_na_vrcholy[1].pop()
    j = stupen_na_vrcholy[max_stupen].pop()
    print( i, j )
    stupen_na_vrcholy[max_stupen-1].append(j)
    if len(stupen_na_vrcholy[max_stupen]) == 0:
        max_stupen -= 1
```

Vyššie popísaná konštrukcia nie je jediná možná, existuje aj viacero iných funkčných postupov. Iné ľahko implementované riešenie vyzerá nasledovne: Po skontrolovaní nutných podmienok už vieme, že riešenie existuje. Aby sme ho zostrojili, rozdelíme si vrcholy na listy (stupňa 1) a vnútorné vrcholy (stupňa väčšieho ako 1). V hotovom strome musia všetky vnútorné vrcholy tvoriť jeden súvislý podgraf. Môžeme teda hľadaný strom zostrojiť tak, že najskôr spojíme vnútorné vrcholy a potom k nim pridáme listy.
No a ľahko nahliadneme, že vnútorné vrcholy môžeme do stromu pospájať ako len chceme, vždy to bude fungovať. Totiž počet „voľných miest, ktoré nám ostanú pre listy“ vôbec nezávisí od toho, akým spôsobom vnútorné vrcholy prepojíme: celkový počet „koncov mostov“ na vnútorných vrcholoch je konštantný a pri prepájaní s inými vnútornými vrcholmi ich vždy dokopy spotrebujeme rovnako. Môžeme preto vnútorné vrcholy prepojiť tým najjednoduchším spôsobom: uložiť ich do radu a pospájať susedné. (Toto s nimi vždy vieme spraviť, keďže každý z nich má stupeň aspoň 2.)

Listing programu (Python)

```
import sys

N = int( input() )
D = [ int(_) for _ in input().split() ]

if min(D) < 1 or max(D) > N-1 or sum(D) != 2*N-2:
    print('neexistuje')
    sys.exit()

vnutorne = [ n for n in range(N) if D[n] > 1 ]
listy    = [ n for n in range(N) if D[n] == 1 ]

for i in range( len(vnutorne)-1 ):
    print( vnutorne[i], vnutorne[i+1] )
    D[ vnutorne[i] ] -= 1
    D[ vnutorne[i+1] ] -= 1

for i in range( len(vnutorne) ):
    while D[ vnutorne[i] ] > 0:
        print( vnutorne[i], listy.pop() )
        D[ vnutorne[i] ] -= 1
```

A-I-4 Exaktné exponenciálne algoritmy

Podúloha A: dva konflikty medzi zvieratami

Predstavme si konflikty medzi zvieratami ako hrany grafu. Ak má každé zviera nanajvýš dva konflikty, má každý vrchol nášho grafu stupeň najviac 2. To ale znamená, že jeho komponenty súvislosti sú len izolované vrcholy, cesty a cykly. A pre každý z nich vieme úlohu o najväčšej nezávislej množine riešiť pažravo:

- Izolované vrcholy (zvieratá bez konfliktov) zoberieme všetky.
- Z cesty dĺžky d zjavne vieme zobrať nanajvýš $\lceil d/2 \rceil$ zvierat.
- Z cyklu dĺžky d zjavne vieme zobrať nanajvýš $\lfloor d/2 \rfloor$ zvierat.[1] (V oboch prípadoch to vieme naozaj dosiahnuť tak, že idúc po ceste/cykle zoberieme každé druhé zviera.)

Podúloha B: lepší algoritmus pre nezávislú množinu

Ak algoritmom z podúlohy A vyriešime vstupy, kde má každé zviera nanajvýš dva konflikty, vieme, že zviera z, ktoré vyberieme v kroku 2 „lepšieho algoritmu 1“ má aspoň tri konflikty. Keď sa teda algoritmus v kroku 4 rekurzívne zavolá na všetky zvieratá okrem z a $N(z)$, bude sa volať na vstup s nanajvýš $n - 4$ zvieratami.
Z vety o časovej zložitosti rekurzie dostávame, že časová zložitosť takto upraveného algoritmu je $\hat{O}(\alpha^n)$, kde α je kladný reálny koreň rovnice $x^4 - x^3 - 1 = 0$.
Pomocou vhodného nástroja (napr. Wolfram Alpha) vieme zistiť, že $\alpha \approx 1.3803$. Dostávame teda algoritmus s časovou zložitosťou $\hat{O}(1.3803^n)$ – čiže efektívnejší, ako oba algoritmy zo študijného textu.

Podúloha C: veža zo škatúľ

Existuje viacero rôznych prístupov, ktoré vedú k rôznym polynómom v časovej zložitosti. Asi najefektívnejšie a najjednoduchšie na implementáciu je riešenie, ktoré vežu stavia zhora dole – teda novú škatuľu budeme vždy vkladať na spodok už existujúcej veže.
Pri takomto riešení nás vlastne pre každú podmnožinu škatúľ zaujíma len jediný bit informácie: dá sa zo všetkých týchto škatúľ postaviť veža? Všimnite si, že akonáhle vieme, ktoré škatule tvoria vežu, je jednoznačne určené aj to, aká je vysoká (súčet výšok škatúľ, ktoré ju tvoria), aj to, aká je ťažká (súčet ich hmotností).
No a keď si kladieme otázku, či sa z danej sady škatúľ dá postaviť veža, jednoducho vyskúšame všetky možnosti pre to, ktorá z škatúľ bude na jej spodku. Do úvahy pripadajú len tie škatule, ktorých nosnosť je aspoň taká ako celková hmotnosť všetkých ostatných škatúľ. Pre každú takúto škatuľu sa rekurzívne pozrieme, či vieme zo zvyšných škatúľ postaviť platnú vežu, ktorá bude stáť na nej. Ak sa nám to niekedy podarí, riešenie existuje, ak nie, tak nie.
Nasleduje iteratívna implementácia tohto riešenia. Jej časová zložitosť je $O(n2^n)$.

Listing programu (C++)

```
// parametre: Vysky, hMotnosti a Nosnosti skatul
int najvyssia_veza(const vector<int> &V, const vector<int> &M, const vector<int> &N) {
    int n = V.size();
    int maximalna_vyska = 0;

    // spravime si pole kde si pre kazdu podmnozinu pamatame ci sa z nej da spravit veza
    vector<bool> moze_veza(1<<n, false);
    moze_veza[0] = true;

    // postupne prejdeme vsetky neprazdne podmnoziny v takom poradi, aby sme v okamihu,
    // kedy spracuvame mnozinu S, mali uz spracovane vsetky jej podmnoziny
    for (int podmnozina=1; podmnozina<(1<<n); ++podmnozina) {
        int celkova_vyska = 0, celkova_hmotnost = 0;
        for (int i=0; i<n; ++i) if (podmnozina & 1<<i) {
            celkova_vyska += V[i];
            celkova_hmotnost += M[i];
        }
        // vyskusame vsetky moznosti ktora z skatul je na spodku veze
        for (int i=0; i<n; ++i) if (podmnozina & 1<<i) {
            if (N[i] >= celkova_hmotnost - M[i] && moze_veza[podmnozina ^ 1<<i]) {
```

[1] Symboly $\lceil \cdot \rceil$ a $\lfloor \cdot \rfloor$ označujú hornú a dolnú celú časť čísla.

```
                moze_veza[podmnozina] = true;
                maximalna_vyska = max( maximalna_vyska, celkova_vyska );
            }
        }
    }
    return maximalna_vyska;
}
```

Všimnite si, že pri implementácii sme použili techniku z posledného odseku študijného textu: podmnožiny škatúľ si reprezentujeme ako prirodzené čísla od 0 po $2^n - 1$. Keď chceme vedieť, či sa škatuľa i nachádza v konkrétnej podmmožine, pozrieme sa, či má príslušné číslo nastavený i-ty bit.

TRIDSIATY PIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2019