69. ročník Matematické olympiády – 2019/2020

Řešení úloh školního kola kategorie P

P-I-1 Schodiště

K určení počtu různých cest nahoru po schodišti použijeme dynamické programování. Postupně pro každý schod budeme počítat, kolika způsoby se na něj dokážeme dostat. Při tom budeme využívat dříve vypočítané údaje. Tento počet způsobů pro schod i označíme p_i

Na schod 0 (což je zem před začátkem schodiště) se dostaneme jedním způsobem – tak, že nic neuděláme. Položíme tedy $p_0 = 1$.

Uvažujme nyní nějaký vyšší schod i. Všechny možné způsoby, kterými se lze dostat na schod i, můžeme rozdělit do několika disjunktních skupin podle toho, ze kterého schodu j jsme udělali poslední krok na schod i. Ze země na nějaký konkrétní schod j se dostaneme přesně p_j způsoby. Platí proto, že p_i je rovno součtu hodnot p_j pro ty schody j, z nichž dokážeme udělat krok na schod i – tedy pro takové schody, pro které platí $j < i$ a $(h_{j+1} + \cdots + h_i) \leq d$.

Dobrá implementace tohoto řešení má časovou složitost v nejhorším případě kvadratickou vzhledem k počtu schodů. Za takové řešení jste mohli získat 6 bodů, případně dokonce 8, pokud jste hledání vyhovujících j začali od $j = i - 1$ směrem dolů a přerušili jste ho, jakmile rozdíl výšek i-tého a j-tého schodu byl už příliš velký.

Vzorové řešení má lineární časovou složitost vzhledem k počtu schodů, tedy časovou složitost $O(n)$. Předchozí řešení zlepšíme pomocí jednoduchého nástroje: prefixových součtů.

Máme-li nějakou posloupnost $a_0, \ldots, a_{n-1}$, můžeme v lineárním čase nalézt novou posloupnost $b_0, \ldots, b_n$ definovanou následovně: $b_0 = 0$ a $\forall i > 0 : b_i = b_{i-1} + a_{i-1}$. Potom platí, že každé b_i je rovno součtu prvních i členů posloupnosti a.

Prefixové součty nejprve použijeme na zadanou posloupnost výšek schodů h_i a vytvoříme si tím novou posloupnost v_i: „nadmořskou výšku“ jednotlivých schodů. Podmínku „dokážeme udělat krok ze schodu j na schod i“ nyní můžeme ověřit v konstantním čase: stačí se podívat, zda $v_i - v_j \leq d$.

Druhým krokem našeho řešení bude, že pro každý schod i najdeme nejnižší schod m_i, z něhož ještě dokážeme udělat krok na schod i. Tyto údaje m_i získáme také v lineárním čase, a to díky jednoduchému pozorování, že hodnoty m_i jsou neklesající. Když například ze schodu 2 nedokážeme udělat krok na schod 7, jistě ho nebudeme moci udělat ani na vyšší schod 8.

Na začátku schodiště víme, že $m_1 = 0$: ze země můžeme udělat krok na první schod. Postavíme nyní Adama na schod 1 a Bětku na schod 0 a budeme opakovat následující proces:

- Adam se posune o jeden schod nahoru. Toto je nový schod i.
- Dokud Bětka stojí na schodu, z něhož nedokáže udělat krok k Adamovi, posune se o jeden schod nahoru. Tímto způsobem Bětka postupně vyjde ze schodu m_{i-1} na schod m_i. Hodnotu m_i si zaznamenáme.

Celý tento proces můžeme odsimulovat v lineárním čase a postupně tak zjistíme všechny hodnoty m_i. Pokud vám není zřejmé, proč má simulace uvedeného procesu lineární časovou složitost, všimněte si, že Adam udělá dohromady přesně $n-1$ kroků a Bětka udělá dohromady nejvýše $n-1$ kroků.

Poslední úpravou původního kvadratického řešení bude, že si zároveň s hodnotami p_i budeme průběžně počítat také jejich prefixové součty s_i.

Výpočet konkrétní hodnoty p_i bude nyní vypadat takto: Nechť $j = m_i$. Víme, že $p_i = p_j + \cdots + p_{i-1}$. Pomocí již spočítaných prefixových součtů posloupnosti p dokážeme tento součet vypočítat v konstantním čase: $p_i = s_i - s_j$. Následně už jenom dopočítáme, že $s_{i+1} = s_i + p_i$ a můžeme přejít na další schod.

Na závěr už jenom podotkneme, že hodnoty p_i mohou růst až exponenciálně. Aby nebylo zapotřebí implementovat aritmetiku velkých čísel, přidali jsme do zadání úlohy větu, že nás z čísla p_n zajímá pouze zbytek po dělení číslem $10^9 + 7$. Tento zbytek zjevně zjistíme tak, že všechny mezivýsledky budeme počítat modulo 10^9+7.

```
#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;

    // vypočítáme výšky, v nichž jsou jednotlivé schody
    vector<long long> V = {0};
    for (int h : H) V.push_back( V.back()+h );

    // pro každý schod vypočítáme první schod, z něhož na něj můžeme udělat krok
    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);
    }

    // postupně počítáme odpovědi a zároveň jejich prefixové součty
    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;
}
```

P-I-2 Nová bankovka

Částečný počet bodů za tuto úlohu bylo možné získat různými způsoby. První sadu testovacích dat stačilo vyřešit hrubou silou – vyzkoušením všech možných způsobů placení. Druhou sadu bylo možné řešit níže popsaným hladovým algoritmem bez vylepšení pomocí Tvrzení 1 či 2. První tři sady lze vyřešit algoritmem, který pro každou částku od 1 do $\max t_i$ zjistí optimální způsob placení pomocí dynamického programování.

Nejprve dokážeme jedno tvrzení o původní sadě platidel. Toto tvrzení platí pro české koruny i pro eura a je zcela intuitivní – naše sady platidel byly navrženy tak, aby se s nimi pohodlně pracovalo. Přesný důkaz ale bude vyžadovat důslednou argumentaci.

Kdybychom neměli žádné nové platidlo, mohli bychom používat jednoduchý hladový algoritmus: Chceme-li nějakou částku zaplatit co nejmenším počtem platidel, stačí vždy použít nejvyšší platidlo, jehož hodnota nepřekročí částku, kterou je ještě třeba zaplatit.

Důkaz tvrzení provedeme odzadu. Mince s hodnotou 1, 2 a 5 nazveme *malé mince.* K zaplacení částky od 1 do 9 tolarů lze použít pouze malé mince. Snadno ověříme, že pro každou z těchto částek najde hladový algoritmus optimální řešení.

Naopak, když zaplatíme jakoukoliv částku optimálním počtem platidel, potom použité malé mince mají v součtu hodnotu menší než 10. Proč tomu tak je? V opačném případě bychom mohli mezi použitými malými mincemi vybrat podmnožinu s hodnotou přesně 10 nebo 11 a místo těchto mincí by pak bylo lepší použít jednu minci s hodnotou 10, resp. nahradit tuto podmnožinu dvojicí mincí $10 + 1$.

Tato dvě pozorování nám dohromady říkají, že ať platíme jakoukoliv částku optimálním způsobem, malými mincemi vždy zaplatíme přesně její poslední cifru. Přitom víme, že to hladový algoritmus udělá vždy optimálně. Nyní můžeme zapomenout na existenci malých mincí, vynulovat poslední cifru částky, kterou platíme, a celou úvahu zopakovat pro další řád.

Takto postupně dostaneme, že hladový algoritmus funguje i pro desítky, stovky, tisíce a desetitisíce tolarů. Dokončení důkazu pro řády od statisíců výše už přenecháme na čtenáře.

Uvažujme nyní, co se změní zavedením nového platidla s nominální hodnotou x. Kdybychom věděli, kolikrát máme při placení částky t použít bankovku x, bylo by to snadné – zbytek částky zaplatíme původní sadou platidel a můžeme tedy použít jednoduchý hladový algoritmus.

My to sice nevíme, ale je zde snadná pomoc: vyzkoušíme „všechny“ možnosti a vybereme z nich tu nejlepší. Při tomto zkoušení možností se samozřejmě vyplatí ještě trochu přemýšlet: například pro $x = 3$ si nemůžeme dovolit zkoušet více než 300 miliónů možností. Pomůže nám jedno z následujících tvrzení:

Tvrzení 1. Pro zadaná omezení platí, že bankovku s hodnotou x vždy použijeme méně než 50 000-krát.

Důkaz. Jestliže $x > 50\,000$, potom ji použijeme méně než 50 000-krát, protože $50\,000^2 > 2 \cdot 10^9 \geq \max t_i$ (zaplatili bychom určitě více než požadovanou částku). Jestliže $x < 50\,000$, nikdy se nevyplatí použít 50 000 kusů bankovky x, neboť místo nich je lepší použít x kusů bankovky s hodnotou 50 000.

Tvrzení 2. Pro zadaná omezené platí, že bankovku s hodnotou x vždy použijeme méně než 40 014-krát.

Důkaz. Libovolnou částku do $2 \cdot 10^9$ dokážeme bez použití bankovky x zaplatit pomocí méně než 40 014 kusů platidel. Kdybychom bankovku x použili vícekrát, měli bychom už příliš mnoho kusů platidel a nejednalo by se tak o optimální řešení.

Implementace řešení je jednoduchá: vyzkoušíme všechny smysluplné možnosti pro počet použitých bankovek s hodnotou x a pro každou z nich spustíme hladový algoritmus na zaplacení zbytku částky zbývajícími typy platidel. Mezi všemi takto získanými řešeními vybereme nejlepší.

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

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

for t in T:
    nejlepsi = hladove(t)
    for kolik_x in range(0, nejlepsi):
        if kolik_x * x > t:
            break
        toto = kolik_x + hladove(t - kolik_x * x)
        nejlepsi = min(nejlepsi, toto)
    print(nejlepsi)
```

P-I-3 Rekonstrukce mapy

V našem řešení budeme používat terminologii z teorie grafů: království je *strom,* ostrovy a mosty jsou jeho *vrcholy a hrany,* počet mostů vedoucích z ostrova nazýváme *stupeň vrcholu.*

Pro $n = 1$ máme jeden vrchol a ten musí mít stupeň 0. Nadále budeme předpokládat, že $n > 1$.

Jelikož strom má $n-1$ hran, každý vrchol musí mít stupeň nejvýše $n-1$. Strom je souvislý, proto každý vrchol musí mít stupeň alespoň 1. Každá hrana přispívá ke stupni dvou vrcholů, takže součet stupňů všech vrcholů musí být přesně $2(n-1)$. Pokud některá z těchto podmínek není splněna, řešení neexistuje.

Ukážeme, že výše uvedená trojice podmínek je nejen nutná, ale zároveň i postačující. Jinými slovy řečeno, jsou-li všechny tři uvedené podmínky splněny, potom strom se zadanými stupni vrcholů vždy existuje. Tvrzení dokážeme matematickou indukcí podle počtu vrcholů stromu.

Pro $n = 2$ existuje jediná posloupnost stupňů splňující všechny podmínky: $(1, 1)$. Pro ni skutečně existuje strom: dva vrcholy spojené hranou.

Mějme nyní posloupnost délky n. Zjevně musí existovat nějaké i takové, že $d_i = 1$ (kdyby všechny stupně byly alespoň 2, byl by jejich součet příliš velký). Nechť j je index odpovídající nejvyšší hodnotě d_j. Vytvoříme novou posloupnost d' z d tak, že vynecháme d_i a o 1 snížíme hodnotu d_j. Nová posloupnost d' splňuje všechny podmínky (rozmyslete si, proč) a protože je kratší, podle indukčního předpokladu jí skutečně odpovídá nějaký strom. Strom pro původní posloupnost nyní sestrojíme tak, že přidáme nový list a připojíme ho hranou k vrcholu odpovídajícímu původní hodnotě d_j.

Uvedený důkaz nám zároveň přímo ukazuje algoritmus, jak hledaný strom sestrojit.

Chceme-li dosáhnout optimální časové složitosti (lineární vzhledem k počtu vrcholů stromu), potřebujeme si ještě rozmyslet, jak šikovně hledat indexy i a j v každé iteraci. V našem řešení to uděláme tak, že si vrcholy rozdělíme na $n - 1$ hromádek podle jejich stupně. Za vrchol i vždy vezmeme libovolný vrchol stupně 1 a za vrchol j libovolný vrchol s aktuálně maximálním stupněm. Maximální stupeň se bude během výpočtu jedině zmenšovat a pokaždé se sníží nejvýše o 1, takže každou iteraci algoritmu skutečně zvládneme v konstantním čase.

```
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+1, j+1 )
    stupen_na_vrcholy[max_stupen-1].append(j)
    if len(stupen_na_vrcholy[max_stupen]) == 0:
        max_stupen -= 1
```

P-I-4 Exaktní exponenciální algoritmy

Část A: dva konflikty mezi zvířaty

Představme si konflikty mezi zvířaty jako hrany grafu. Má-li každé zvíře nejvýše dva konflikty, má každý vrchol našeho grafu stupeň nejvýše 2. To ale znamená, že jeho komponenty souvislosti jsou jenom izolované vrcholy, cesty a cykly. A pro každý z nich dokážeme úlohu o největší nezávislé množině vyřešit hladově:

- Izolované vrcholy (zvířata bez konfliktů) vezmeme všechna.
- Z cesty délky d zjevně můžeme vybrat nejvýše $\lceil d/2 \rceil$ zvířat.
- Z cyklu délky d zjevně můžeme vybrat nejvýše $\lfloor d/2 \rfloor$ zvířat.* (V obou případech to uděláme tak, že jdeme po cestě/cyklu a vezmeme každé druhé zvíře.)

Část B: lepší algoritmus pro nezávislou množinu

Jestliže algoritmem z části A vyřešíme vstupy, v nichž má každé zvíře nejvýše dva konflikty, víme, že zvíře z vybrané v kroku 2 „lepšího algoritmu 1“ má alespoň tři konflikty. Když se tedy algoritmus v kroku 4 rekurzivně zavolá na všechna zvířata kromě z a $N(z)$, bude se volat na vstup s nejvýše $n - 4$ zvířaty.

Z věty o časové složitosti rekurze dostaneme, že časová složitost takto upraveného algoritmu je $\hat{\mathcal{O}}(\alpha^n)$, kde α je kladný reálný kořen rovnice $x^4 - x^3 - 1 = 0$.

Pomocí vhodného nástroje (například Wolfram Alpha) můžeme zjistit, že $\alpha \approx 1.3803$. Dostáváme tedy algoritmus s časovou složitostí $\hat{\mathcal{O}}(1.3803^n)$ – což je efektivnější, než oba algoritmy ze studijního textu.

Část C: věž z krabic

Existuje více různých přístupů, které vedou k různým polynomům v časové složitosti. Asi nejefektivnější a zároveň nejjednodušší na implementaci je řešení, které staví věž shora dolů – novou krabici budeme vždy vkládat dospodu již existující věže.

Při takovém řešení nás vlastně pro každou podmnožinu krabic zajímá jenom jediný bit informace: je možné ze všech těchto krabic postavit věž? Všimněte si, že jakmile víme, které krabice tvoří věž, pak už je jednoznačně dáno, jak je věž vysoká (součet výšek krabic ve věži) a jak je těžká (součet jejich hmotností).

Když si klademe otázku, zda lze z dané sady krabic postavit věž, jednoduše vyzkoušíme všechny možnosti, která z krabic bude umístěna úplně dole. V úvahu připadají jenom ty krabice, jejichž nosnost je rovna alespoň celkové hmotnosti všech ostatních krabic. Pro každou takovou krabici se rekurzivně podíváme, zda dokážeme ze zbývajících krabic postavit věž, která na ní bude stát. Pokud se nám to někdy podaří, řešení existuje, v opačném případě neexistuje.

Následuje iterační implementace tohoto řešení. Její časová složitost je $O(n2^n)$.

```
// parametry: Výšky, hMotnosti a Nosnosti krabic
int nejvyssi_vez(const vector<int> &V, const vector<int> &M, const vector<int> &N) {
    int n = V.size();
    int maximalni_vyska = 0;
```

* Symboly $\lceil \cdot \rceil$ a $\lfloor \cdot \rfloor$ označují horní a dolní celou část čísla.

```
    // vytvoříme pole, kde si pro každou podmnožinu pamatujeme,
    // zda z ní lze postavit věž
    vector<bool> lze_vez(1<<n, false);
    lze_vez[0] = true;

    // postupně projdeme všechny neprázdné podmnožiny v takovém pořadí, abychom
    // při zpracování množiny S měli již zpracované všechny její podmnožiny
    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];
        }
        // vyzkoušíme všechny možnosti, která z krabic je na spodku věže
        for (int i=0; i<n; ++i) if (podmnozina & 1<<i) {
            if (N[i] >= celkova_hmotnost - M[i] && lze_vez[podmnozina ^ 1<<i]) {
                lze_vez[podmnozina] = true;
                maximalni_vyska = max(maximalni_vyska, celkova_vyska);
            }
        }
    }
    return maximalni_vyska;
}
```

Všimněte si, že při implementaci jsme použili techniku z posledního odstavce studijního textu: podmnožiny krabic reprezentujeme pomocí přirozených čísel od 0 do $2^n - 1$. Když chceme zjistit, zda se krabice i nachází v konkrétní podmnožině, podíváme se, zda má příslušné číslo nastaveno i-tý bit.