67. ročník Matematické olympiády – 2017/2018

Řešení úloh krajského kola kategorie P

P-II-1 Trénink

Úloha má více různých řešení „hrubou silou“. Na tři body stačilo samostatně vyzkoušet každý dostatečně dlouhý úsek. Čtyři body jste dostali, pokud jste předchozí řešení trochu vylepšili: když už víme, kde má maximum úsek $a_i, \ldots, a_j$, snadno v konstantním čase zjistíme, kde má maximum úsek $a_i, \ldots, a_{j+1}$.

Pátý bod se dal získat za jednoduché, ale velmi užitečné pozorování: Namísto úseků délky *aspoň* k stačí uvažovat úseky délky *přesně* k. Když totiž víme, že nějaká hodnota x je maximem nějakého dlouhého úseku, pak zjevně totéž x zůstane maximem, i když tento úsek z libovolného konce zkrátíme (samozřejmě tak, aby x stále v úseku zůstalo).

Úseků délky k je jen $n - k + 1$. Budeme-li každý z nich kontrolovat zvlášť, dostaneme řešení s časovou složitostí $\mathcal{O}(k(n-k+1))$, což můžeme shora odhadnout jako $\mathcal{O}(nk)$.

```
N, K = [ int(_) for _ in input().split() ]
A    = [ int(_) for _ in input().split() ]

mozna_maxima = set()
for zacatek in range(N-K+1): mozna_maxima.add( max( A[zacatek:zacatek+K] ) )
print( len(mozna_maxima) )
```

Skoro-vzorové řešení

Existuje více způsobů, jak lze úlohu vyřešit s časovou složitostí $\mathcal{O}(n \log n)$ nebo $\mathcal{O}(n \log k)$. Za kteroukoliv z těchto časových složitostí se dalo získat 8 bodů.

Jedno možné řešení je založeno na pozorování, že když jsme právě zpracovali úsek $a_i, \ldots, a_{i+k-1}$ a chceme zpracovat úsek $a_{i+1}, \ldots, a_{i+k}$, potřebujeme provést pouze dvě změny: ze zpracovaného úseku odstranit hodnotu a_i a naopak do něj přidat novou hodnotu a_{i+k}.

Jestliže si prvky aktuálního úseku budeme udržovat v uspořádané množině, dokážeme v čase $\mathcal{O}(\log k)$ přidat novou hodnotu, odstranit odebíranou hodnotu i zjistit, jakou hodnotu má aktuální maximum.

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

int main() {
    int N, K; cin >> N >> K;
    vector<int> A(N); for (int &a:A) cin >> a;

    set<int> aktualni_usek( A.begin(), A.begin()+K );
    int aktualni_max = *aktualni_usek.rbegin();
    int pocet_maxim = 1;
```

```
    for (int zacatek=1; zacatek+K<=N; ++zacatek) {
        aktualni_usek.erase( A[zacatek-1] );
        aktualni_usek.insert( A[zacatek+K-1] );
        int nove_max = *aktualni_usek.rbegin();
        if (nove_max != aktualni_max) {
            aktualni_max = nove_max;
            ++pocet_maxim;
        }
    }
    cout << pocet_maxim << endl;
}
```

Jiné šikovné řešení používá intervalový strom. Vytvoříme si prázdné pole velikosti n a nad tímto polem si postavíme intervalový strom. V každém vrcholu stromu si budeme pamatovat, zda už je někde pod ním nějaké neprázdné políčko, a pokud ano, jaký je nejmenší a největší index takových políček.

Do pole nyní budeme postupně zapisovat na správná místa čísla naší posloupnosti, ale ne zleva doprava, nýbrž *od největšího po nejmenší.*

Pokaždé, když chceme přidat nějaké číslo x na nějaký index i, nejprve se našeho intervalového stromu zeptáme na dvě otázky: kde je nejbližší už zaplněné políčko nalevo a napravo od indexu i? Obě otázky umíme pomocí údajů uložených ve vrcholech intervalového stromu zodpovědět v logaritmickém čase. Odpovědi na tyto otázky udávají levý a pravý okraj nejdelšího úseku, jehož maximem je právě přidávaná hodnota x. Stačí už jenom zkontrolovat, zda je tento úsek dostatečně dlouhý.

Vzorové řešení

Vzorové řešení bude mít optimální časovou složitost $\mathcal{O}(n)$, je tedy lineární vzhledem k velikosti vstupu. Bude založeno na stejném pozorování jako předcházející řešení: abychom zjistili, zda je konkrétní hodnota x maximem dostatečně dlouhého úseku, stačí nalézt nejbližší větší hodnotu nalevo a napravo od x.

Ukážeme, jak můžeme při jednom průchodu polem zleva doprava v lineárním čase určit ke každému prvku pole nejbližší větší prvek ležící nalevo od něho. Stejnou funkci potom použijeme ještě jednou na zrcadlově převrácenou posloupnost, abychom našli také nejbližší větší prvky ležící napravo od každého prvku.

Budeme postupně zleva doprava zpracovávat prvky naší posloupnosti. V každém okamžiku si budeme pamatovat množinu těch prvků, které ještě mohou být „nejbližším větším prvkem nalevo“ pro nějaký prvek, který přijde v budoucnosti.

Klíčové je toto pozorování: Jakmile zpracujeme prvek s hodnotou x, můžeme zapomenout všechny prvky, které jsou nalevo od něho a mají menší hodnotu. Tyto prvky už v budoucnosti nikdy nebudou pro nikoho nejbližším větším prvkem nalevo, neboť x je zastíní.

Příklad: představme si, že jsme už zpracovali posloupnost 100, 14, 74, 39, 40, 27 a 12. V tomto okamžiku si stačí pamatovat pouze tyto hodnoty (a jejich indexy): 100, 74, 40, 27 a 12. Kdyby následujícím prvkem posloupnosti bylo číslo 47, mohli bychom po jeho zpracování zapomenout další tři prvky a pamatovali bychom si už jen hodnoty 100, 74 a 47.

Zpracování jednoho prvku x bude tedy vypadat následovně:

- Zapomeneme všechny prvky, které jsou menší než x.
- Nejmenší z prvků, které si ještě stále pamatujeme, je nejbližším větším prvkem nalevo od x.
- Přidáme x mezi prvky, které si pamatujeme.

Nyní už jen stačí všimnout si, že prvky, které si pamatujeme, budou vždy automaticky uspořádány od největšího po nejmenší – pokaždé totiž zapomínáme od nejmenšího a v okamžiku, když přidáváme nový prvek, je tento nejmenším mezi právě pamatovanými prvky. Nepotřebujeme proto žádnou složitou datovou strukturu – na uložení aktuálně pamatovaných prvků nám stačí obyčejný zásobník.

Odhad časové složitosti: Každý prvek jednou zpracujeme a vložíme do zásobníku, a každý prvek nejvýše jednou ze zásobníku vyhodíme. Proto dohromady vykonáme při průchodu celou posloupností $\mathcal{O}(n)$ operací.

```
N, K = [ int(_) for _ in input().split() ]
A    = [ int(_) for _ in input().split() ]

def odzadu(A): return list(A)[::-1]

def indexy_vetsich_nalevo(A):
    odpoved = []
    kandidati = []
    for i in range(len(A)):
        # vyhodime prilis male kandidaty
        while kandidati != [] and A[kandidati[-1]] < A[i]:
            kandidati.pop()
        # zapíšeme si index nejbližšího většího než A[i]:
        odpoved.append( -1 if kandidati == [] else kandidati[-1] )
        # přidáme nového kandidáta
        kandidati.append(i)
    return odpoved

nalevo  = indexy_vetsich_nalevo(A)
napravo = odzadu( N-1-x for x in indexy_vetsich_nalevo( odzadu(A) ) )

muze_byt_maximum = [ (napravo[i] - nalevo[i] - 1 >= K) for i in range(N) ]
print( sum(muze_byt_maximum) )
```

Alternativní vzorová řešení

Existují také jiná lineární řešení této úlohy. Jedno dostaneme tak, že upravíme řešení, které postupně zpracovávalo úseky a pamatovalo si jejich obsah v uspořádané množině (setu). Místo setu totiž můžeme šikovně využít tzv. oboustrannou frontu (deque). Detaily tohoto řešení ponecháme čtenářům jako domácí úlohu. Napovíme vám ještě, že klíčové je opět si všimnout, že když právě do úseku přibyla nová hodnota x, pak staré hodnoty menší než x už nikdy nebudou maximem zkoumaného úseku. Můžeme je proto klidně zapomenout.

Jiné lineární řešení dostaneme tak, že si vstupní posloupnost nakrájíme na kousky délky k a v každém kousku si spočítáme prefixová a sufixová maxima. Z takto získaných údajů už dokážeme vypočítat maximum libovolného úseku délky k v konstantním čase.

P-II-2 Telenovela II

Na začátku řešení stejně jako v domácím kole ošetříme to, že je třeba vidět první a poslední díl telenovely: najdeme první výskyt prvního dílu, poslední výskyt posledního dílu, ty označíme jako zhlédnuté a od této chvíle se už budeme zabývat pouze úsekem mezi nimi, včetně nich (nebo zjistíme, že úloha nemá řešení).

Po této úpravě už nemusíme nijak zvlášť ošetřovat první a poslední epizodu. Zjevně totiž existuje optimální řešení, které obě tyto epizody použije, takže když určíme délku celkově nejlepšího řešení, bude to zároveň délka nejlepšího řešení, v němž Ondra začne první epizodou a skončí poslední epizodou.

Hlavní myšlenkou našeho vzorového řešení bude následující pozorování: Bez ohledu na to, zda je optimálním řešením rostoucí posloupnost nebo posloupnost s jednou výjimkou, optimální řešení jistě můžeme získat tak, že zpracovávanou posloupnost vhodně „rozstřihneme“ na dvě části a v každé z nich zvlášť najdeme nejdelší rostoucí podposloupnost. Budeme tedy postupovat následovně:

1. Postupně pro každé k zjistíme, jaká nejdelší rostoucí podposloupnost končí k-tým prvkem naší posloupnosti.
2. Hodnoty získané v kroku 1 projdeme zleva doprava a spočítáme jejich prefixová maxima. Pro každé k tak získáme délku s_k nejdelší rostoucí podposloupnosti, kterou lze vybrat z prvních k prvků.
3. Postupně pro každé k zjistíme, jaká nejdelší rostoucí podposloupnost začíná k-tým prvkem naší posloupnosti. (V tomto kroku procházíme zadanou posloupnost zprava doleva a děláme totéž jako v kroku 1, jenom hledáme klesající posloupnost namísto rostoucí.)
4. Hodnoty získané v kroku 3 projdeme zprava doleva a spočítáme jejich sufixová maxima. Pro každé ℓ tak získáme délku t_ℓ nejdelší rostoucí podposloupnosti, kterou lze vybrat z posledních ℓ prvků vstupu.
5. Optimálním řešením je nejvyšší z hodnot $s_k + t_{n-k}$. V lineárním čase vyzkoušíme všechna možná k a vybereme to nejlepší.

Jedinou netriviální částí popsaného řešení je krok 1. (A také krok 3, který je jeho kopií. V naší implementaci na oba šikovně použijeme stejnou funkci.) Algoritmus řešení tohoto kroku jsme si už ale ukázali ve vzorovém řešení domácího kola.

Připomeňme si, jak jsme v domácím kole hledali nejdelší rostoucí podposloupnost. Udržovali jsme si hodnoty c_j s následujícím významem: když se podíváme na všechny možné rostoucí j-prvkové podposloupnosti v již zpracovaných datech a vezmeme poslední prvek každé z nich, nejmenší z takto získaných hodnot bude právě c_j. Speciálně budeme mít $c_0 = -\infty$ (za posloupnost délky 0 se dá přidat cokoliv) a $c_j = +\infty$, jestliže ve zpracovaných datech ještě žádná rostoucí j-prvková podposloupnost neexistuje.

Ukázali jsme si, že vždy po přečtení a zpracování dalšího prvku posloupnosti se změní nejvýše jedna z hodnot c_j. Nyní už jen stačí uvědomit si, že index této hodnoty nám určuje délku nejdelší rostoucí podposloupnosti, která končí právě zpracovaným prvkem.

Příklad: Jestliže jsme již zpracovali $(10, 3, 8, 6, 9, 4, 6, 22, 8, 5)$, budeme mít $c_1 = 3$, $c_2 = 4$, $c_3 = 5$, $c_4 = 8$, a $c_5 = +\infty$. Všimněte si, že různé hodnoty c_j mohou odpovídat různým podposloupnostem. Například v této chvíli nejlepší podposloupností délky 3 je $(3, 4, 5)$, zatímco pro délku 4 to je $(3, 4, 6, 8)$.

Kdyby následujícím prvkem postupnosti byl $x = 7$, první hodnota c_j, která je větší nebo rovná x, je c_4. Tuto hodnotu tedy zmenšíme z 8 na 7. Zároveň jsme zjistili, že nejdelší rostoucí podposloupnost končící tímto x má délku právě 4. Kdybychom namísto toho měli $x = 100$, měnili bychom hodnotu c_5, takže jsme právě zjistili, že nejdelší posloupnost končící právě zpracovávanou hodnotou 100 má délku 5. Pokud $x = 5$, nezměnilo by se nic, neboť právě máme $c_3 = 5$. Našli jsme tedy jen nový způsob, jak získat rostoucí posloupnost délky 3 končící číslem 5.

Kroky 1 a 3 umíme takto implementovat v čase $\mathcal{O}(n \log n)$, ostatní kroky v lineárním čase. Celková časová složitost tohoto řešení je proto $\mathcal{O}(n \log n)$.

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

// v čase n log n vypočítáme délku nejdelší rostoucí podposloupnosti
// pro každý prefix A
vector<int> delky_lis(const vector<int> &A) {
    // inicializujeme si C tak, aby C[0]=-inf; v každém okamžiku
    // si budeme pamatovat jen tu část C, která je < inf
    vector<int> C(1, -1);
    vector<int> odpoved;
    unsigned nejvice = 0;
    for (int a : A) {
        // najdeme nejmenší i takové, že C[i] >= a
        unsigned i = upper_bound( C.begin(), C.end(), a-1 ) - C.begin();
        // upravíme hodnotu C[i] na a
        if (i == C.size()) C.push_back(a); else C[i] = a;
        nejvice = max( nejvice, i );
        odpoved.push_back(nejvice);
    }
    return odpoved;
}

int main() {
    int N, E; cin >> N >> E;
    vector<int> epizody(N); for (int n=0; n<N; ++n) cin >> epizody[n];

    // najdeme první vysílání první epizody a poslední vysílání poslední epizody
    int prvni = 0, posledni = N-1;
    while (prvni < N && epizody[prvni] != 1) ++prvni;
    while (posledni >= 0 && epizody[posledni] != E) --posledni;

    // když se nestíhají obě nebo některou vůbec nevysílají, řešení neexistuje
    if (posledni < prvni) { cout << -1 << endl; return 0; }

    // sestrojíme si posloupnost, v níž budeme skutečně hledat
    vector<int> zustalo( epizody.begin()+prvni, epizody.begin()+posledni+1 );

    // zjistíme optimální délku rostoucí podposloupnosti pro každý prefix
    vector<int> S = delky_lis(zustalo);

    // obrátíme posloupnost, přečíslujeme epizody naopak a použijeme tutéž funkci
```

```
    // na nalezení optimální délky rostoucí podposloupnosti pro každý sufix
    reverse( zustalo.begin(), zustalo.end() );
    for (int &e : zustalo) e = E+1-e;
    vector<int> T = delky_lis(zustalo);

    // určíme nejlepší způsob, jak naši posloupnost rozdělit na prefix a sufix
    int odpoved = 0;
    for (int i=1; i<zustalo.size(); ++i)
        odpoved = max( odpoved, S[i-1]+T[zustalo.size()-i] );
    cout << odpoved << endl;
}
```

Alternativní pomalejší řešení

Ukážeme si ještě řešení se snadnější implementací, ale s časovou složitostí $\mathcal{O}(n^2)$. Upravíme pomalejší řešení z domácího kola.

Stejně jako v domácím kole označme b_i délku nejdelší rostoucí podposloupnosti, jejíž poslední prvek je na indexu i. Navíc nyní označme c_i délku nejdelší rostoucí podposloupnosti s hledanou vlastností (tedy rostoucí až na nejvýše jednu výjimku), jejíž poslední prvek je na indexu i.

Hodnoty b_i už umíme počítat. Hodnoty c_i počítáme následovně: Je-li poslední prvek na indexu i a předposlední na indexu $j < i$, jsou dvě možnosti. Jestliže $a_j < a_i$, můžeme vzít nejdelší posloupnost s nejvýše jednou výjimkou, která končí na indexu j, a za ni přidat prvek na indexu i. Pokud $a_j \geq a_i$, pak tím, že za prvek na indexu j přidáme prvek na indexu i, vyrobíme výjimku. Od začátku po prvek na indexu j proto můžeme vzít jenom posloupnost, která je čistě rostoucí.

Dohromady dostáváme následující řešení:

```
// vstup máme uložen v proměnné N a v poli hodnot A[0..N-1]
vector<int> B(N), C(N);

for (int i=0; i<N; ++i) {
    B[i] = C[i] = 1;
    for (int j=0; j<i; ++j) if (A[j] < A[i]) B[i] = max( B[i], 1+B[j] );
    for (int j=0; j<i; ++j) {
        if (A[j] < A[i]) C[i] = max( C[i], 1+C[j] );
        else             C[i] = max( C[i], 1+B[j] );
    }
}
```

P-II-3 Vaření

Tato úloha úzce souvisí s úlohami o nejkratších cestách v grafu. Všechny postupy, které si ukážeme, budou přímým využitím grafových algoritmů.

Optimální vaření je acyklické

Představme si, že už známe pro každé jídlo nejnižší cenu jedné jeho porce.

Jestliže si jídla podle této ceny uspořádáme, zjevně bude platit, že při optimální přípravě konkrétního jídla lze použít jako ingredience pouze jídla „nalevo“ od něho, tedy jídla s nižší cenou.

Kdyby nám někdo správné pořadí jídel (to podle optimální ceny) prozradil, mohli bychom úlohu vyřešit snadno: šli bychom ve správném pořadí zleva doprava a počítali bychom optimální ceny.

Určení optimální ceny pro konkrétní jídlo by bylo jednoduché: buď ho přímo koupíme, nebo ho uvaříme podle některého receptu. V této chvíli už známe optimální ceny obou jeho ingrediencí a z nich snadno spočítáme optimální cenu právě připravovaného jídla.

V tomto okamžiku už dokážeme implementovat první korektní, i když zoufale pomalé řešení: vyzkoušíme všech $n!$ možných pořadí jídel a pro každé z nich použijeme výše popsaný postup. Nejlepší způsob, jak uvařit jídlo číslo 1, je potom minimem ze všech $n!$ možností.

Bellmanův-Fordův algoritmus

Nyní si ukážeme jednoduchý polynomiální algoritmus, který řeší naši úlohu. Pro každé jídlo i bude b_i označovat nejnižší cenu, za jakou ho aktuálně umíme uvařit. Začneme tím, že každé b_i nastavíme na cenu c_i, za niž můžeme příslušné jídlo koupit v obchodě.

Výpočet bude až překvapivě jednoduchý: postupně $(n-1)$-krát projdeme celý seznam receptů a pro každý z nich zkontrolujeme, zda pomocí něho můžeme zlepšit hodnotu b pro to jídlo, které se tímto receptem vytváří.

Máme-li tedy recept, který z jídel s_i a t_i uvaří jídlo u_i, podíváme se, zda $b_{s_i} + b_{t_i}$ není menší než dosavadní hodnota b_{u_i}. Pokud ano, hodnotu b_{u_i} snížíme na právě nalezenou lepší cenu.

Je zjevné, že pro n jídel a r receptů má tento algoritmus časovou složitost $\mathcal{O}(nr)$. Opravdu ale funguje?

Dokážeme, že když algoritmus skončí, všechny hodnoty b_i odpovídají skutečně nejnižším cenám, za něž lze získat jednotlivá jídla.

Opět si představme, že nám už někdo seřadil jídla do pořadí $p_0, \ldots, p_{n-1}$ podle optimální ceny, za niž je dokážeme získat.

Zjevně platí $b_{p_0} = c_{p_0}$, tedy nejlevnější jídlo je nejlepší rovnou koupit. Už na začátku výpočtu (po inicializaci hodnot b_i na c_i) proto známe optimální cenu jídla p_0.

Nyní stačí uvědomit si, že každým průchodem všemi recepty určíme výslednou cenu aspoň jednoho dalšího jídla v našem optimálním pořadí: po prvním průchodu seznamem receptů budeme znát optimální cenu pro p_1, po druhém i pro p_2, po třetím i pro p_3, a tak dále.

Důvod je stejný jako v úvaze z předchozí části řešení: Nechť platí, že po i průchodech seznamem receptů známe optimální ceny pro jídla $p_0, \ldots, p_i$. Když budeme ještě jednou procházet všemi recepty, vyzkoušíme (kromě jiného) všechny způsoby, jak z už optimálně vyřešených jídel uvařit jídlo p_{i+1}, takže určitě najdeme také optimální řešení pro toto jídlo.

Může se samozřejmě stát, že při jednom průchodu recepty najdeme optimální řešení pro více jídel (možná dokonce rovnou pro všechna jídla). Místo přesně $n-1$ průchodů seznamem receptů proto můžeme zformulovat náš algoritmus také

následovně: procházej opakovaně seznamem receptů tak dlouho, až se při některém průchodu žádná z hodnot b_i nezmění. Takto implementovaný Bellmanův-Fordův algoritmus bude mít stále v nejhorším případě časovou složitost $\Theta(nr)$, ale v praxi bude často o něco rychlejší.

Dijkstrův algoritmus

Je asi dost zřejmé, co je pomalého na předchozím algoritmu: v každé „fázi" znovu a znovu procházíme úplně všechny recepty, přičemž mnohé nám nijak nepomohou – některé proto, že už jsme je pro aktuální vstupy použili, některé zase proto, že pro jejich vstupy ještě neznáme optimální ceny.

Mnohem lepší by bylo, kdybychom znali správné pořadí jídel podle optimální ceny, potom by stačilo každý recept vyzkoušet jen jednou.

Na první pohled to zní jako začarovaný kruh – to pořadí se přece snažíme zjistit. Trik spočívá v tom, že to pořadí nepotřebujeme znát hned celé. Během výpočtu programu ho budeme postupně sestrojovat.

Při výpočtu tohoto nového algoritmu budeme jídla postupně označovat jako hotová. V okamžiku, když nějaké jídlo označíme za hotové, budeme si jistí, že jeho hodnota b_i už odpovídá optimální ceně jeho výroby.

Jídla, která ještě nejsou hotová, budeme nazývat kandidáti. Pro každého kandidáta i bude platit, že b_i je minimum z dvou možností: jeho nákupní ceny c_i a nejlevnějšího způsobu, jak lze jídlo i vyrobit jen pomocí již hotových jídel.

Na začátku výpočtu budou všechna jídla kandidáty. Protože ještě nemáme hotová žádná jídla, jediným povoleným způsobem přípravy je přímé zakoupení, takže pro každé jídlo platí $b_i = c_i$.

Hlavní myšlenkou algoritmu bude následující pozorování: Kandidáta, jehož hodnota b_i je nejmenší mezi všemi kandidáty, můžeme prohlásit za hotového.

Proč tomu tak je? Proto, že jeho cenu už nemůžeme nijak zlepšit. Za cenu menší než současné b_i nedokážeme ani uvařit nic jiného z už hotových jídel, ani žádné jiné jídlo koupit.

Nyní tedy známe další jídlo, které můžeme prohlásit za hotové. Když se tak stane, potřebujeme zabezpečit, aby opět platilo, že pro ostatní kandidáty známe nejlevnější způsob přípravy pomocí hotových jídel. Tím, že nám přibylo jedno hotové jídlo, přibyly nám možná také nějaké nové levnější způsoby, jak uvařit jiné kandidáty. Na tyto postupy se musíme podívat. Naštěstí jich není až tak mnoho – stačí prohlédnout ty recepty, které obsahují jako surovinu to jídlo, které jsme právě prohlásili za hotové.

V popsaném algoritmu tedy každé jídlo právě jednou prohlásíme za hotové a každý recept právě dvakrát zpracujeme (vždy, když prohlásíme za hotovou jednu ze dvou ingrediencí, které se v něm používají).

Potřebujeme ještě umět efektivně provádět dvě věci: Za prvé, potřebujeme k danému jídlu projít všechny recepty, v nichž se používá. To je snadné, už při čtení vstupu si můžeme recepty roztřídit do samostatných seznamů pro každé jídlo.

Za druhé, v každém kroku výpočtu potřebujeme určit, které jídlo mezi kandidáty můžeme prohlásit za hotové.

Toto lze snadno řešit hrubou silou: v každém kole výpočtu projdeme všechny kandidáty a určíme, který z nich má nejmenší hodnotu b_i. Jelikož kol, stejně jako jídel, bude n a v každém z nich strávíme hledáním $\mathcal{O}(n)$ času, bude mít toto řešení časovou složitost $\mathcal{O}(n^2 + r) = O(n^2)$.

Šikovnější je ale použít lepší datovou strukturu. Kandidáty si například můžeme udržovat uspořádané podle aktuální hodnoty b_i. Přesněji, budeme mít uspořádanou množinu, v níž budou uloženy záznamy ve tvaru (b_i, i). Při této implementaci najdeme nejlevnějšího kandidáta v čase $\mathcal{O}(\log n)$ – je jím aktuální minimum. Naopak se nám trochu zpomalí zpracování receptu. Když totiž nějakému jídlu chceme snížit jeho hodnotu b_i, musíme to provést i s jeho záznamem v naší uspořádané množině. To se nejsnáze provede tak, že smažeme starý záznam a přidáme nový. Taková implementace má časovou složitost $\mathcal{O}((n + r) \log n)$.

Existuje ještě lepší implementace s časovou složitostí $\mathcal{O}(n \log n + r)$, ta ale používá pokročilé datové struktury a získané zlepšení už není příliš zajímavé. Stejně dobré časové složitosti, jakou má naše vzorové řešení, lze dosáhnout také implementací pomocí prioritní fronty. Ta se od popsaného postupu liší tím, že záznamy nemažeme. Když nějakou hodnotu b_i zlepšíme, jednoduše do prioritní fronty vložíme i nový, lepší záznam. Při tomto řešení může prioritní fronta narůst až na $\mathcal{O}(r)$ záznamů, to nám ale časovou složitost nepokazí.

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

typedef struct { int par, vystup; } recept;
typedef struct { long long cena; int jidlo; } zaznam;

bool operator< (const zaznam &A, const zaznam &B) {
    return A.cena < B.cena || (A.cena == B.cena && A.jidlo < B.jidlo);
}

int main() {
    int N, M;
    cin >> N >> M;
    vector<long long> nakup(N);
    for (int n=0; n<N; ++n) cin >> nakup[n];
    vector< vector<recept> > kucharka(N);
    for (int m=0; m<M; ++m) {
        int vysledek, surovina1, surovina2;
        cin >> vysledek >> surovina1 >> surovina2;
        --vysledek; --surovina1; --surovina2;
        kucharka[surovina1].push_back( {surovina2,vysledek} );
        kucharka[surovina2].push_back( {surovina1,vysledek} );
    }

    set<zaznam> Q;
    for (int n=0; n<N; ++n) Q.insert( {nakup[n],n} );
    vector<long long> nej = nakup;

    while (!Q.empty()) {
        int mam = Q.begin()->jidlo;
```

```
        Q.erase( Q.begin() );
        for (const recept &r : kucharka[mam]) {
            if (nej[mam] + nej[r.par] < nej[r.vystup]) {
                Q.erase( { nej[r.vystup], r.vystup } );
                nej[r.vystup] = nej[mam] + nej[r.par];
                Q.insert( { nej[r.vystup], r.vystup } );
            }
        }
    }

    cout << nej[0] << endl;
}
```

P-II-4 Stavebnice funkcí

Postupně si ukážeme řešení jednotlivých podúloh.

Část A: umocňování

Podobně jako sčítání bylo opakovaným použitím následovníka a násobení opakovaným použitím sčítání, umocňování je opakovaným použitím násobení. Výpočet x^y si můžeme představit tak, že začneme od hodnoty 1 a tu postupně y-krát vynásobíme hodnotou x.

Vidíme ale jeden drobný rozdíl. Sčítání i násobení bylo komutativní, takže například u násobení bylo jedno, zda x-krát sečteme y nebo zda y-krát sečteme x. V případě umocňování už na pořadí parametrů záleží.

Chtěli bychom zde nějaký postup opakovat y-krát, Cyklovač ale umí jako počet opakovaní použít jedině první parametr, nikoliv druhý. Budeme proto postupovat podobně, jako když jsme sestrojovali odčítání: nejprve si vytvoříme pomocnou funkci *wop*, která provádí umocňování, ale má opačné pořadí parametrů – tedy $\forall x, y : wop(x, y) = y^x$.

Funkci *wop* chceme vytvořit pomocí Cyklovače. Pojďme tedy zjistit, jaké dvě funkce f a g do něj musíme vložit, aby nám z něj vypadlo právě takovéto umocňování. Když do Cyklovače vložíme unární funkci f a ternární funkci g, dostaneme následující binární funkci:

```
def wop(x,y):
    tmp = f(y)
    for i = 0 to x-1:
        tmp = g(i,y,tmp)
    return tmp
```

Pro $x = 0$ tato funkce vrátí hodnotu $f(y)$. Protože cokoliv umocněné na nultou je jedna, potřebujeme jako f použít unární konstantní funkci, která vždy vrátí hodnotu 1, tedy funkci k_1^1. Program se tím upravil následovně:

```
def mul(x,y):
    tmp = 1
    for i = 0 to x-1:
        tmp = g(i,y,tmp)
    return tmp
```

Nyní bychom chtěli, aby každá iterace cyklu vynásobila pomocnou proměnnou *tmp* proměnnou y. Proto jako g použijeme funkci *tří* proměnných, která na výstup vrátí součin druhého a třetího vstupu. To je *skoro,* ale ne úplně, funkce *mul*. My už ale umíme pomocí Kompozitoru upravit počet a pořadí vstupů funkce. Hledanou funkci g sestrojíme například takto: $g \equiv K[v_2^3, v_3^3, mul]$.

Dohromady tak dostáváme, že $wop \equiv C[\, k_1^1,\, K[v_2^3, v_3^3, mul]\,]$.

Zbývá už jen vyměnit pořadí parametrů: $pow \equiv K[v_2^2, v_1^2, wop]$.

Část B: signum

Funkci signum nejsnáze sestrojíme drobným trikem pomocí Cyklovače. Proměnnou *tmp* na začátku nastavíme na 0 a v každém cyklu ji potom nastavíme na 1. Jestliže se tedy neprovedla ani jedna iterace cyklu, vrátíme nulu, jinak vrátíme jedničku.

Pro korektní použití Cyklovače potřebujeme funkci f *s aritou 0,* která vždy vrátí nulu, a funkci g *s aritou 2,* která vždy vrátí jedničku. Platí tedy: $sgn \equiv C[z, k_1^2]$.

Část C: predikát „větší nebo rovno“

Jak můžeme zjistit, zda je x ostře větší než y? Od x odečteme y a podíváme se, zda nám ještě něco zbylo.

Jak můžeme zjistit, zda je x větší nebo rovno y? Protože x i y jsou nezáporná celá čísla, stačí k číslu x přičíst jedničku a potom se podívat, zda je nové x větší než y.

Postupujme tedy následovně:

- Pomocná funkce $f_1 \equiv K[v_1^2, s]$ má dva vstupy, vezme první z nich, přičte k němu jedničku a získanou hodnotu dá na výstup.
- Pomocná funkce $f_2 \equiv K[f_1, v_2^2, sub]$ má dva vstupy. K prvnímu z nich přičte jedničku a potom od něho odečte druhý z nich.
- Už víme, že f_2 vrátí kladnou hodnotu právě tehdy, když její první vstup byl větší nebo roven druhému vstupu. Zbývá provést jednu drobnou úpravu – místo kladné hodnoty chceme vracet na výstup hodnotu 1. To je ale triviální, stačí na výstup funkce f_2 použít funkci signum sestrojenou v části B. Predikát *geq* tedy sestrojíme takto: $geq \equiv K[f_2, sgn]$.

Část D: větvení

V této části úlohy jsme měli vzít neznámé, ale již sestrojené funkce f_0, f_1 a predikát g a měli jsme ukázat, jak z nich sestrojit funkci h definovanou předpisem:

$$h(x_1, \ldots, x_n) = \begin{cases} f_0(x_1, \ldots, x_n) & \text{jestliže } g(x_1, \ldots, x_n) = 0 \\ f_1(x_1, \ldots, x_n) & \text{jestliže } g(x_1, \ldots, x_n) = 1 \end{cases}$$

Začneme tím, že si definujeme funkci *not*, která bude počítat logickou negaci – tedy pro nulu vrátí jedničku a pro jedničku (a také pro jakýkoliv jiný kladný vstup) vrátí nulu. Tato funkce je velmi podobná funkci signum a můžeme ji definovat přesně stejným způsobem: $not \equiv C[k_1^0, k_0^2]$.

Nyní už můžeme přímo sestrojit funkci h. Pomůžeme si jednoduchým trikem. Výběr z dvou hodnot zařídíme tak, že obě sečteme, ale tu z nich, kterou právě nechceme, vynásobíme nulou. Jinými slovy, uvědomíme si, že platí následující vztah:

$$h(x_1,\ldots,x_n) = not(g(x_1,\ldots,x_n)) \cdot f_0(x_1,\ldots,x_n) + g(x_1,\ldots,x_n) \cdot f_1(x_1,\ldots,x_n).$$

Pokud totiž pro nějaká $x_1,\ldots,x_n$ predikát g vrátí nulu, zjednoduší se náš výraz na

$$h(x_1,\ldots,x_n) = 1 \cdot f_0(x_1,\ldots,x_n) + 0 \cdot f_1(x_1,\ldots,x_n) = f_0(x_1,\ldots,x_n),$$

a to je přesně to, co jsme chtěli. Když g vrátí jedničku, díky násobení nulou zmizí zase druhý člen:

$$h(x_1,\ldots,x_n) = 0 \cdot f_0(x_1,\ldots,x_n) + 1 \cdot f_1(x_1,\ldots,x_n) = f_1(x_1,\ldots,x_n).$$

Hledaná funkce h bude tedy součtem dvou funkcí a každá z nich bude součinem dvou funkcí s aritou n. Formálně můžeme zapsat:

$$h \equiv K[\ \underbrace{K[\,K[g, not], f_0, mul\,]}_{\text{toto je } not(g)\cdot f_0},\ K[\,g, f_1, mul\,],\ add\ ].$$