B-II-1 Preteky v sánkovaní

Našou úlohou je nájsť také poradie n súťažiacich, že každé z m očakávaní, čo o nich máme bude naplnené, alebo musíme zistiť, že také poradie neexistuje.
Aby sa nám nad úlohou lepšie rozmýšľalo, mali by sme si zvoliť vhodnú reprezentáciu. V tomto prípade sa ponúka použitie orientovaného grafu. Každý zo súťažiacich bude predstavovať jeden vrchol a každé očakávanie bude orientovaná hrana. Ak očakávame, že x_i predbehne y_i, do grafu pridáme hranu vedúcu z vrchola y_i do vrchola x_i.
Keď sa pozrieme na takýto graf, pomerne jednoducho pomenujeme *nutnú podmienku* existencie vhodného poradia – náš graf nemôže obsahovať žiaden cyklus. Cyklus v orientovanom grafe je postupnosť vrholov, medzi ktorými vedú hrany: $v_1 \to v_2 \to \cdots \to v_k \to v_1$. To ale znamená, že očakávame, že v_2 predbehne v_1, v_3 predbehne v_2, $\ldots$ v_k predbehne v_{k-1} a v_1 predbehne v_k. To je však spor. Nech si zvolíme ľubovoľné poradie, jeden z týchto vrcholov v ňom bude posledný a teda určite nepredbehne vrchol, ktorý je v cykle pred ním.

Toto je síce dôležité pozorovanie, k riešeniu úlohy nám to však veľmi nepomáha. Nevieme totiž ako by sme cykly v grafe hľadali a navyše, nevieme, či je táto podmienka aj *dostačujúca* – čo ak by pre graf, ktorý neobsahuje cyklus napriek tomu neexistovalo vhodné poradie.
Skúsime teda navrhnúť algoritmus, ktorý nám nielenže pomôže nájsť výslednú postupnosť, ale zároveň bude slúžiť ako dôkaz toho, že naša nutná podmienka je aj dostačujúca.

Poďme vytvoriť nejaké vhodné poradie. Ktorý vrchol (súťažiaci) môže byť na poslednom mieste? Určite to nemôže byť vrchol, do ktorého vedie nejaká hrana. To by totiž znamenalo, že očakávame, že tento vrchol predbehne nejaký iný a tým pádom nemôže byť posledný. Vhodní kandidáti sú teda všetky vrcholy, do ktorých nevedie žiadna hrana.
Musí aspoň takýto vrchol existovať? Musí. Totiž keby do každého vrcholu viedla nejaká hrana, tak sa môžeme začať po našom grafe pohybovať. Začneme, kde len chceme, a potom si opakovane vyberieme hranu, ktorá do aktuálneho vrcholu vedie, a prejdeme po nej (proti smeru šípky) do iného vrcholu. Z každého vrcholu sa budeme vedieť pohnúť ďalej, tento proces teda nikdy neskončí. Časom sa nám teda musí stať, že do niektorého vrcholu prídeme druhýkrát. To ale znamená, že sme práve prešli po nejakom cykle!
Vieme teda, že ak graf nemá cyklus, nutne v ňom existujú nejakí kandidáti na posledného – vrcholy, do ktorých nevedie žiadna hrana.
Závisí, ktorý z týchto vrcholov prehlásime za posledný v celkovom poradí? Nie. Tieto vrcholy spolu nesúvisia a navyše neexistuje očakávanie, ktoré by sme zaradením ľubovoľného z nich nesplnili. Vyberme si preto ľubovoľný z nich, ktorý nazveme x a zaradíme ho na posledné miesto nášho poradia. Vrchol x by sme následne chceli odstrániť z grafu. Ale síce do neho žiadne hrany nevedú, nejaké z neho môžu vychádzať, keď očakávame, že iné vrcholy predbehnú vrchol x. Uvedomme si však, že aj tieto hrany sú už nepodstatné. Vrchol x je totiž na konci poradia a teda ho predbehnú *všetky* ostávajúce vrcholy. Tieto očakávania teda určite budú naplnené a my si ich v grafe nemusíme ďalej pamätať.
Po odstránení vrcholu x a všetkých z neho vedúcich hrán nám ostane menší graf. S ním však potrebujeme riešiť rovnakú úlohu – nájsť poradie vrcholov spĺňajúce všetky hranové očakávania. A keď toto poradie nájdeme a na koniec pridáme x, dostaneme poradie pre pôvodnú úlohu.
Opäť teda budeme hľadať ľubovoľný vrchol, do ktorého nevchádzajú žiadne hrany. Uvedomme si, že odstránením x mohli pribudnúť nové vrcholy s touto vlastnosťou – tie, do ktorých vtedy už išla hrana iba z vrcholu x.

Náš algoritmus teda opakovane vyberie vrchol, do ktorého nevchádza hrana, tento vrchol umiestni ako ďalší v poradí a tento vrchol (a jemu prislúchajúce hrany) z grafu vymaže. Algoritmus zastane iba v jednom z dvoch prípadov. V prvom postupne vyberie a odstráni všetky vrcholy grafu a tým pádom vytvorí celkové poradie, ktoré muselo spĺňať všetky očakávania. Takéto poradie stačí vypísať.
V druhom prípade však algoritmus zastane preto, že síce sú ešte v grafe nejaké vrcholy, žiaden z nich nemá nula vchádzajúcich hrán. To ale inými slovami znamená, že do každého vrcholu zostávajúceho grafu vchádza aspoň jedna hrana. A bez ohľadu na to, ktorý vrchol by sme vybrali ako ďalší v poradí, táto jedna hrana by bola očakávanie, ktoré by sme už nevedeli naplniť. V tomto prípade teda algoritmus prehlási, že požadované poradie neexistuje.
Môžete si navyše rozmyslieť, že tento prípad nastane naozaj iba vtedy, ak sa v grafe nachádza cyklus.

Ostáva naše riešenie *efektívne* implementovať. Pri tom môžeme postupovať tak, že najprv navrhneme ľubovoľný fungujúci spôsob a ten potom vylepšíme, keď sa zamyslíme, či nejaké úkony nerobíme zbytočne alebo opakovane. Jadrom úlohy je opakovane hľadať taký vrchol, do ktorého nevchádza žiadna hrana. Najjednoduchšie, čo môžeme spraviť, je najprv si pre každý vrchol spočítať, koľko hrán doňho vchádza. Vytvoríme si pole `stupen[]`, kde na pozícii i bude počet týchto hrán pre vrchol i. Následne prejdeme postupne každú hranu $y_i \to x_i$ a zväčšíme hodnotu `stupen[`x_i`]` o jedna. Potom môžeme týmto poľom prejsť a nájsť taký vrchol, ktorý sme ešte nezaradili do poradia a má stupeň 0. Tento vrchol vymažeme tak, že odstránime všetky hrany, ktoré z neho vedú a vrchol označíme za použitý.

Tu sa ponúka hneď niekoľko vylepšení, ktorých aplikácia povedie k optimálnej časovej zložitosti. Prvým je, že ak chceme odstraňovať všetky hrany vedúce z nejakého vrcholu v, najlepšie je mať ich zapamätané ako zoznam susedov, aby sme nemuseli prechádzať všetky vrcholy v celom grafe, ale iba naozaj tie, do ktorých vedú hrany vychádzajúce z v.

Zaujímavejšie je vytváranie poľa `stupen[]`. Počítať ho totiž zakaždým nanovo prejdením všetkých hrán je totiž zdĺhavé. Predstavme si, že máme správne vyplnené pole `stupen[]` (napríklad sme ho na začiatku *raz* pracne vypočítali) a odstraňujeme vrchol v. Ako sa zmení toto pole odstránením vrchola v?

S vrcholom v zmiznú všetky z neho vedúce hrany. A iba neprítomnosť týchto hrán spôsobí, že pole `stupen[]` bude iné. Konkrétne, ak máme hranu $v \to w$, tak vrchol w bude mať o jedna menší stupeň. Namiesto toho, aby sme prechádzali všetky hrany a počítali pole `stupen[]` nanovo, prejdeme iba hrany, ktoré vychádzajú z v a zmenšíme príslušné políčka.

Posledné zlepšenie sa týka toho, ako hľadáme vrchol so stupňom 0. Triviálne riešenie je vždy prejsť poľom `stupen[]`, to však trvá dlho. Namiesto toho si uvedomíme, že keď vrchol dosiahne 0 vchádzajúcich hrán, tento počet sa už nezmení. Spravíme si množinu vrcholov, ktoré majú stupeň 0 a vždy budeme vyberať z tejto množiny. Na začiatku, keď vytvoríme pole `stupen[]`, prejdeme ho celé a všetky vrcholy s hodnotou 0 vložíme do našej množiny. Čas ušetríme vždy, keď namiesto prechodu celého poľa iba vyberieme prvok z množiny.

Ako však do množiny dostať nové vrcholy? Ako budeme vedieť, že nejaký vrchol má už stupeň 0? Uvedomme si, že klesnúť na 0 môže stupeň vrcholu iba keď odstraňujeme hranu a teda upravujeme `stupen[]`. Takže vždy, keď odstránime hranu $v \to w$ a `stupen[w]` klesne z hodnoty 1 na hodnotu 0, pridáme vrchol w do našej množiny.

A to je celé riešenie. Aká je výsledná časová zložitosť? Na začiatku musíme prejsť všetky hrany, zapísať si čísla do poľa `stupen[]`, prejsť týmto poľom a vybrať všetky vrcholy so stupňom 0. Toto bude trvať $O(n+m)$. Následne najviac n krát vyberieme ľubovoľný prvok z našej množiny a odstránimi hrany, ktoré z neho vedú, čím upravíme hodnoty `stupen[]` a prípadne vložíme nové vrcholy do našej množiny. V rámci celého algoritmu však každú hranu odstránime najviac raz. To znamená, že časová zložitosť odstraňovania bude dokopy $O(m)$. Výsledná časová zložitosť je $O(n+m)$, pamäťová je rovnaká.

Listing programu (Python)

```
n, m = map(int, input().split())

hrany = [[] for _ in range(n)]
stupen = [0 for _ in range(n)]
mnozina = []
vysledok = []

for _ in range(m):
    x, y = map(int, input().split())
    x, y = x - 1, y - 1
    hrany[y].append(x)
    stupen[x] += 1

for i in range(n):
    if stupen[i] == 0:
        mnozina.append(i)

while len(mnozina) != 0:
    v = mnozina.pop()
    vysledok.append(v + 1)
    for w in hrany[v]:
        stupen[w] -= 1
        if stupen[w] == 0:
            mnozina.append(w)

if len(vysledok) != n: print('Miska sa myli')
else: print(*reversed(vysledok))
```

(Výsledné poradie, ktoré sme v tejto úlohe zostrojili, zvykneme nazývať *topologickým usporiadaním* daného orientovaného grafu. Existujú aj iné rovnako efektívne algoritmy ako ten vzorový, napríklad sa dá na riešenie tejto úlohy upraviť aj prehľadávanie do hĺbky.)

B-II-2 Fúzna záhradka

Na začiatok spravme niekoľko jednoduchých, ale dôležitých pozorovaní. Je zrejmé, že ak má Peťo reaktor, ktorý má hodinovú spotrebu vyššiu alebo rovnú ako je jeho maximálny výkon, takýto reaktor do zátišia nikdy nebude chcieť zaradiť, lebo by jeho použitím nezískal energiu. Predpokladajme teda, že všetky reaktory spĺňajú $s_i < v_i$.

V niektorých úlohách, v ktorých sa snažíme nájsť najlepšiu možnú hodnotu, čo je v našom prípade hľadanie najlepšieho e je oveľa ľahšie *overiť* či ju nejaký konkrétny výsledok spĺňa, ako tento výsledok nájsť. Pozrime sa preto, či v našom prípade vieme pre **zvolenú hodnotu** e vypočítať celkový výstup zátišia.
Ak si zoberieme reaktor i s hodnotami v_i a s_i tak si môžeme uvedomiť, že tento reaktor vieme dať do zátišia iba v prípade, že $v_i \geq e$, inak by sme ho totiž nevedeli nastaviť na požadovaný výkon. A zároveň, takýto reaktor vyrába energiu iba ak platí, že $e > s_i$. No a ak chceme maximalizovať množstvo vyrábanej energiu, určite sa oplatí zobrať každý reaktor, čo nejakú vyrába. Ak teda máme zvolenú hodnotu e, môžeme prejsť postupne všetky reaktory a do zátišia zaradiť tie, pre ktoré platí $v_i \geq e > s_i$.

Vďaka tomuto pozorovaniu sa nám naskytá riešenie, kde vyskúšame všetky možné e, pre ne vypočítame celkový výstup optimálneho zátišia a vyberieme to e, ktoré nám dalo najvyššiu hodnotu.
Ak chceme toto riešenie zrýchliť, oplatí sa zamyslieť nad otázkou, či sa naozaj oplatí skúšať všetky možné e. Nevieme z nich vybrať len niektoré „zaujímavé“?
Predstavme si, že sme si zobrali nejakú hodnotu e a vybrali do zátišia všetky reaktory, ktoré sa pre túto hodnotu oplatia. Pozrime sa na výkony všetkých vybratých reaktorov. Najmenší z nich označíme v_{min}. Vieme, že $v_{min} \geq e$. Predpokladajme, že platí nerovnosť $v_{min} > e$. Čo sa teraz stane, ak zväčšíme hodnotu e na hodnotu v_{min}? Stále budeme môcť použiť tie isté reaktory (všetky, ktoré sme mali vybraté, majú dostatočný výkon) a navyše nám to zvýši celkový energetický výstup.
To ale znamená, že sa nikdy neoplatí vybrať také e, ktoré nie je rovné niektorému výkonu reaktora. Máme teda len n zmysluplných možností na výber hodnoty e a to sú práve všetky hodnoty v_i.

Z týchto pozorovaní dostávame nasledovné riešenie. Za hodnotu e dosadíme postupne všetky hodnoty v_i. Pre každé takto zvolené e vypočítame najväčší možný celkový výstup. To spravíme tak, že o každom reaktore zistíme, koľko by v takomto zátiší produkoval energie. A ako výsledok si zvolíme to e, ktoré nám dalo najvyššiu hodnotu vyprodukovanej energie.
Pre každú z n hodnôt čísla e prejdeme všetkých n reaktorov, dostaneme teda riešenie s časovou zložitosťou $O(n^2)$.

Pre zlepšenie tohto riešenia sa bližšie pozrime na krok, v ktorom zisťujeme celkový výstup pre hodnotu e. Pomohlo by nám, keby sme pri overovaní jednej hodnoty e mohli využiť aspoň časť výpočtu z predchádzajúceho kroku, čo odstráni zbytočné opakovanie.
Zoraďme si všetky reaktory podľa ich maximálneho výkonu od najväčšieho po najmenší a overujme hodnoty e v tomto poradí. Keď bude e najväčšie možné, použiť môžeme iba prvý reaktor, ktorý zaradíme do zátišia. Keď sa posunieme na druhý najväčší výkon, do zátišia nám môže pribudnúť práve reaktor s týmto výkonom. A všetky predchádzajúce reaktory stále môžu byť v zátiší, keďže ich maximálny výkon je väčší a vieme ich teda nastaviť aj na túto hodnotu. Využívame teda, že vždy keď pridávame ďalší reaktor, všetky ktoré sme spracovali predtým sú takisto vhodné.
A celkový výstup budeme počítať tak, že si v premennej `pocet` budeme pamätať koľko reaktorov je v zátiší a v premennej `spotreba` aká je ich celková spotreba. Keď pridáme nový reaktor i, jednoducho zväčšíme `pocet` o 1 a hodnotu `spotreba` o s_i. Celkový výkon potom bude $v_i \cdot pocet - spotreba$ (aktuálna hodnota e je rovná v_i).

Toto riešenie však ešte nie je korektné, lebo reaktory nikdy neodstraňuje. Nám sa však v nejakom momente stane, že s_i reaktora, ktorý máme v zátiší, bude väčšie ako aktuálne e a tým pádom tento reaktor prestane

produkovať energiu. Aj tu však poznáme poradie, v akom to bude nastávať. Prvý totiž vypadne reaktor, ktorý má najväčšiu hodnotu s_i a tak to pôjde postupne od najväčších spotrieb po najmenšie.
Môžeme si teda všetky spotreby usporiadať od najväčších po najmenšie a vždy keď zmeníme hodnotu e (a pridáme nový reaktor), tak skontrolujeme, či spotreba nejakých reaktorov nepresiahla e. A keďže platí $v_i > s_i$, tieto reaktory sú započítané v našom zátiší a stačí ich teda odstrániť. Na to zmenšíme o 1 hodnotu `pocet` a o príslušnú spotrebu hodnotu `spotreba`.

Náš algoritmus teda usporiada dve polia – v jednom sú dvojice (výkon, spotreba), v druhom iba spotreba – a potom postupne vyskúša n rôznych hodnôt e, pre každú vypočíta celkový výstup zátišia. Pri tomto výpočte vždy pridá jeden reaktor a niekoľko odoberie. Dokopy však každý reaktor pridá práve raz a najviac raz každý reaktor odoberie. Časová zložitosť tohto kroku je preto $O(n)$. Počas behu algoritmu si zapamätáme, pre ktoré e bol celkový výstup najväčší. Na konci iba prejdeme reaktory a vyberieme tie, ktoré sa pre túto hodnotu oplatí použiť, čím dostaneme požadovaný výstup.
Najpracnejšou časťou nášho riešenia je usporiadanie dvoch polí, celková časová zložitosť je $O(n \log n)$ a pamäťová $O(n)$.

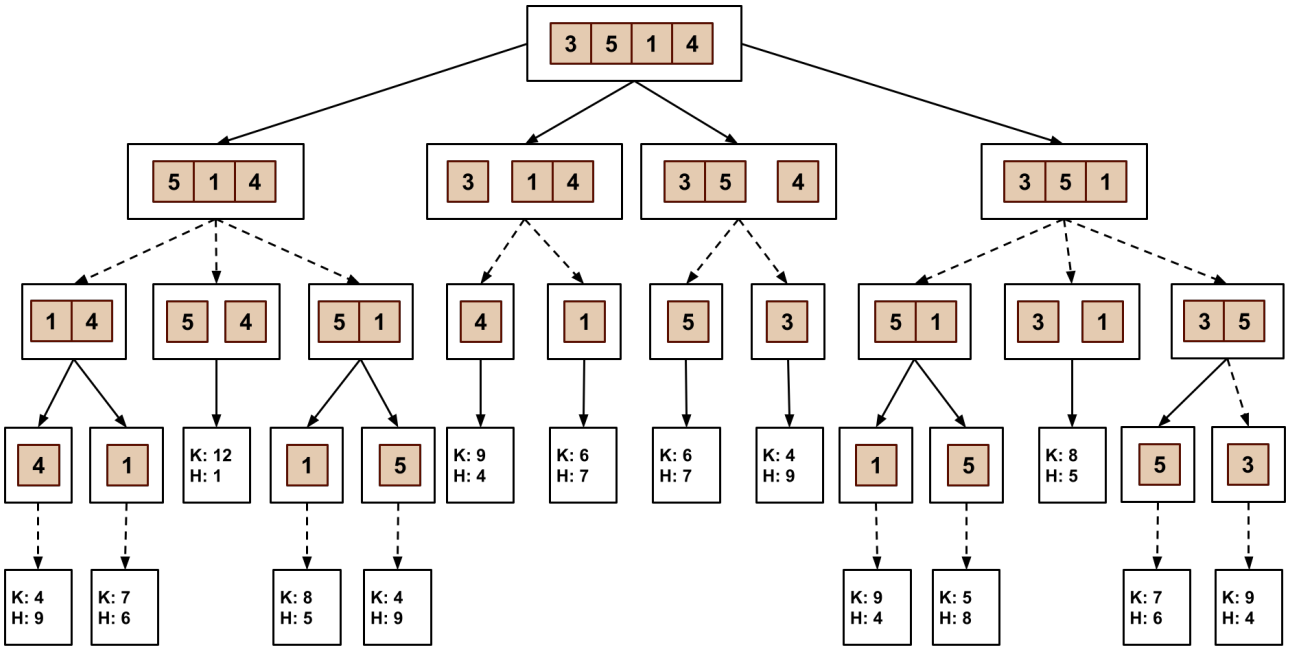
Listing programu (Python)

```
n = int(input())
reaktory = []
vykony = []
spotreby = []

for _ in range(n):
    s, v = map(int, input().split())
    reaktory.append((v, s))
    if v > s:
        vykony.append((v, s))
        spotreby.append(s)

vykony.sort(reverse=True)
spotreby.sort(reverse=True)

pocet = 0
spotreba = 0
pozicia = 0
najlepsi_vystup = -1
najlepsie_e = -1

for i in range(len(vykony)):
    e = vykony[i][0]
    pocet += 1
    spotreba += vykony[i][1]

    while pozicia != len(vykony) and spotreby[pozicia] >= e:
        pocet -= 1
        spotreba -= spotreby[pozicia]
        pozicia += 1

    if najlepsi_vystup < e * pocet - spotreba:
        najlepsi_vystup = e * pocet - spotreba
        najlepsie_e = e

zobrat = []
for i, (v, s) in enumerate(reaktory):
    if v >= najlepsie_e and najlepsie_e > s:
        zobrat.append(i + 1)

print(najlepsie_e)
print(*zobrat)
```

B-II-3 Čokoláda

V tejto úlohe sme hrali *hru*, podobne ako v druhej úlohe domáceho kola a mali sme zistiť, ako hra skončí, ak obaja hráči hrajú *optimálne*.

Začnime tým, že sa pozrieme na všetky možné priebehy hry (nie nutne najlepšie hrané) pre čokoládu $(3, 5, 1, 4)$. Tie si vieme zakresliť do postupne sa rozvetvujúceho *stromu*, kde každá šípka predstavuje jeden možný ťah. Kubíkove ťahy sú označené plnými šípkami, Hodoboxove prerušovanými. Môžete si sami overiť, že na obrázku sú nakreslené naozaj všetky možnosti, ako vedia chlapci hrať.



Ako vie však Kubík (a symetricky aj Hodobox) využiť tento obrázok pri vyberaní svojich ťahov? Môžu sa pozerať, aké možnosti bude mať ich súper, zistiť ktorú z nich si vyberie a tým pádom zistiť aj to, k akému celkovému výsledku to povedie.

Ukážme si to na príklade z obrázku. Ako Kubík zistí, ktorý prvý ťah je najlepší? Mohol by zobrať prvý štvorček s 3 orieškami, čím by Hodoboxovi posunul čokoládu $(5, 1, 4)$. Hodobox má teraz na výber z troch možností. Je však v jeho záujme zjesť štvorček s 1 orieškom? Zjavne nie, lebo v takom prípade by sa šípkou posunul do stavu, v ktorom Kubík zje 12 orieškov a on iba 1. Iný z ťahov ho určite dostane do stavu, kde zje viac orieškov. A keďže Hodobox hrá *optimálne*, tento ťah nespraví. A toto si vie uvedomiť aj Kubík.

Ktorý ťah teda spraví? Ak zje štvorček s 5 orieškami, na výber bude mať opäť Kubík a keďže aj on hrá optimálne, určite si vyberie tú možnosť, kde on zje 7 a Hodobox 6 orieškov. A ak zje štvorček so 4 orieškami, Kubík si určite vyberie možnosť, pri ktorej zje 8 a Hodobox 5 orieškov. Je teda jasné, že Hodobox si bude chcieť vždy zvoliť možnosť, v ktorej zje prvý štvorček s 5 orieškami.

Kubík si teda pohľadom na obrázok vie uvedomiť, že ak v prvom ťahu zje štvorček s 3 orieškami, hra skončí tak, že on zje dokopy 7 a Hodobox 6 orieškov. Rovnakým spôsobom však vie zvážiť všetky jeho ťahy a zistí:

- Ak zjem štvorček s 3 orieškami, hra skončí tak, že ja zjem 7 orieškov a Hodobox 6 orieškov. (Príklad vyššie.)
- Ak zjem štvorček s 5 orieškami, hra skončí tak, že ja zjem 6 orieškov a Hodobox 7 orieškov. (Hodobox mu určite nechá štvorček s 1 orieškom.)
- Ak zjem štvorček s 1 orieškom, hra skončí tak, že ja zjem 4 oriešky a Hodobox 9 orieškov.
- Ak zjem štvorček so 4 orieškami, hra skončí tak, že ja zjem 8 orieškov a Hodobox 5 orieškov. (Ak by Hodobox v druhom ťahu zjedol iný štvorček ako ten s 5 orieškami, vedel by Kubík zjesť až 9 orieškov, preto to Hodobox určite neurobí.)

No a rozhodnutie je jasné. Najlepšie dopadne hra, v ktorej na začiatku zje štvorček so 4 orieškami.

Prvé riešenie: prehľadávanie celého stromu

Ak vie takéto úvahy robiť Kubík, nemal by byť problém implementovať riešenie, ktoré takýto strom vytvorí a vypočíta, ktorý ťah sa oplatí najviac. Predstavme si, že máme funkciu `najviac_orieskov(cokolada)`, kde

`cokolada` je nejaká vhodná reprezentácia pásikov čokolády. Bez ohľadu na to, kto je na ťahu, obaja chlapci sa snažia získať čo najviac orieškov, a práve to bude počítať naša funkcia.
Ak chceme zistiť, koľko najviac orieškov vieme zjesť zo zadanej čokolády, musíme vyskúšať spraviť každý možný ťah, pre ne zistiť koľko orieškov vďaka nim vieme zjesť a vybrať ten ťah, ktorý mal toto číslo najväčšie. Ako však zistíme, koľko orieškov zjeme, keď spravíme nejaký ťah?
Určite zjeme oriešky, ktoré sú v štvorčekoch popísaných týmto ťahom (keďže `cokolada` sa môže skladať z viacerých kúskov, naraz môžeme zjesť aj viac štvorčekov). Ich zjedením sa nám navyše čokoláda rozpadne na ďalšie kusy, ktoré si nazveme `nova_cokolada`.
Túto novú čokoládu má dostať druhý hráč. A jeho cieľom bude takisto spraviť taký ťah, ktorý maximalizuje počet zjedených orieškov. Toto nám však hovorí funkcia `najviac_orieskov()`, výsledok funkcie `najviac_orieskov(nova_cokolada)` je teda práve najväčší možný počet orieškov, ktorý sa z nej dá zjesť. Nuž a my predsa zjeme všetky oriešky, ktoré nezjedol druhý hráč. Ak si teda `oriesky` označíme počet orieškov v `cokolada`, tak pri zvolenom ťahu zjeme `oriesky - najviac_orieskov(nova_cokolada)` orieškov.
Toto vieme zapísať nasledovným pseudokódom:

Listing programu (Python)

```
def najviac_orieskov(cokolada):
    # ak nam uz ziadna cokolada neostala, odpoved je 0
    if cokolada je zjedena:
        return 0

    najlepsie = 0
    oriesky = pocet orieskov v cokolada
    # postupne vyskusame spravit kazdy mozny tah
    for tah in mozne_tahy(cokolada):
        # useky cokolady po vykonani zvoleneho tahu
        nova_cokolada = cokolada po vykonani tahu tah

        najlepsie = max(najlepsie, oriesky - najviac_orieskov(nova_cokolada))

    # vratime vysledok pre najlepsi tah
    return najlepsie
```

Všimnime si, že v každom ťahu ubudne aspoň jeden štvorček čokolády, preto algoritmus zaručene dobehne do konca. Bude mu to však trvať pomerne dlho. Už len rôznych rozdelení čokolády na kúsky je zhruba 2^n. A do tých istých rozkúskovaní môže viesť viacero ciest, čo nám veľkosť nášho stromu ešte zväčší. Potrebujeme preto prísť na nejaké zlepšenie.

Zlepšenie: nepočítajme veci dvakrát

Uvedomme si, že bez ohľadu na to, akým spôsobom sa dostaneme k nejakému rozkúskovaniu čokolády, výpočet od toho momentu je zakaždým rovnaký. Preto môžeme použiť **memoizáciu** (rovnako ako v domácom kole, úlohe 4). Výsledky, ktoré vypočítame si budeme pamätať tak, aby sme pre rozkúskovanie `cokolada` vedeli rýchlo povedať, koľko najviac orieškov z tohto rozkúskovania vieme získať.
Vždy, keď zavoláme funkciu `najviac_orieskov()`, najprv zistíme, či sme už takéto rozkúskovanie `cokolada` nepočítali predtým. Ak áno, iba vrátime zapamätanú hodnotu, ak nie, výpočet pokračuje tak ako predtým. Akurát na jeho konci si výsledok zapamätáme.
Stále môžeme mať 2^n rôznych rozkúskovaní (každý štvorček buď je alebo nie je v rozkúskovaní), teraz však nepočítame žiadne veci zbytočne dvakrát. A ak sa nám podarí zmenšiť počet možných rozkúskovaní, ako to spravíme v poslednej časti, efektivita tohto prístupu bude ešte väčšia.

Vzorové riešenie: kúsky čokolády sú separátne hry

Predstavme si naslednovnú zmenu našej hry. Vždy, keď hráč dostane dva kúsky čokolády, odloží jeden z nich stranou a pokračuje iba s druhým kúskom. Až keď sa ten úplne celý zje, zoberie odložený kúsok a hru pokračuje s ním.
Môžeme vidieť, že nová hra je ekvivalentná hre zo zadania. Tie isté ťahy, ktoré mohli chlapci urobiť v starej hre, môžu robiť aj v zmenenej a naopak. Akurát ich nespravia naraz, ale postupne, podľa toho, s ktorým kúskom čokolády práve hrajú. Kúsky čokolády sú totiž od seba **nezávislé**, to čo sa udeje s jedným nijak neovplyvní druhý.

Táto zmena však výrazne zredukuje počet možných stavov. Namiesto „všetky nezjedené kúsky", ktorých je 2^n, nám stačí „jeden kúsok", ktorých je iba n^2 – každý kúsok je určený začiatkom a koncom.
Môžeme sa teda inšpirovať našim pseudokódom, použiť rekurziu s memoizáciou, akurát sa budeme na každý kúsok čokolády pýtať separátne.
Každý pásik čokolády spracujeme nanajvýš raz. Jeho dĺžka, ktorá je najviac n, určuje počet možných ťahov, ktoré musíme vyskúšať, preto je časová zložitosť $O(n^3)$. Pamäťová zložitosť je kvadratických $O(n^2)$, keďže si musíme pamätať výsledok pre každý možný kúsok čokolády.

Listing programu (Python)

```
memo = {}

# Pasik [z, k) z cokolady
def najviac_orieskov(z, k, cokolada):
    if (z, k) in memo:
        return memo[(z, k)]
    # ak nam ziadna cokolada neostala, odpoved je 0
    if z == k:
        return 0

    najlepsie = 0
    oriesky = sum(cokolada[z:k])
    # hrac na tahu zje i-ty stvorcek cokolady
    for i in range(z, k):
        # cokolada sa rozdeli na dva kusky [z, i) a [i, k)
        najlepsie = max(najlepsie,
                        oriesky - najviac_orieskov(z, i, cokolada) - najviac_orieskov(i + 1, k, cokolada))

    # zapamatame si vysledok pre najlepsi tah
    memo[(z,k)] = najlepsie
    return najlepsie

n = int(input())

cokolada = list(map(int, input().split()))
print(najviac_orieskov(0, n, cokolada))
```

Ešte raz si teda zhrňme, čo vlastne predchádzajúce riešenie robí. Pre každý možný súvislý kúsok čokolády si položíme otázku: „Koľko najviac orieškov viem zjesť, ak začínam hru a držím v ruke práve tento kúsok čokolády?"
Každú takúto otázku zodpovieme rekurzívne, a to tak, že vyskúšame všetky možnosti, ako spraviť prvý ťah. Keď sme si nejakú vybrali, tak presne vieme povedať, čo sa stane: V prvom rade vieme, koľko orieškov sme práve zjedli. No a ak po našom ťahu ostali ešte nejaké kúsky čokolády (pre ktoré je teraz na ťahu náš optimálne hrajúci súper), tak *samostatne pre každý z nich* rekurzívnym volaním našej funkcie zistíme, koľko najviac orieškov vie súper zjesť, a z toho si vypočítame, koľko ich z daného kúsku zjeme my – to sú všetky na ňom, mínus tie, ktoré zje súper.
Dokopy sme si postupne položili $O(n^2)$ rôznych otázok. Počas zodpovedania každej z nich sme postupne vyskúšali $O(n)$ možností, ako v danej situácii spraviť prvý ťah.

B-II-4 Katakomby

Všimnime si, že keď sme v ľubovoľnom maxime našej funkcie, tak smerom doprava vyzerá úplne rovnako ako smerom doľava: v oboch smeroch ideme q krokov dole, potom q hore, a tak ďalej. To isté platí aj pre minimá. Graf našej funkcie je teda v každom z týchto miest nutne súmerný podľa zvislej osi.

Podúloha A: optimálna stratégia

Ukážeme si teraz stratégiu, ktorá sa spýta **najviac tri otázky** a potom bude vedieť odpovedať.
Najskôr sa spýtame sa na hodnoty $f(1)$ a $f(3)$. Ak katakomby spravili medzi súradnicou 1 a 3 krok hore aj dole (v ľubovoľnom poradí), dostaneme dve rovnaké hodnoty. Ak spravili dva kroky tým istým smerom, dostaneme dve hodnoty, ktoré sa líšia o $2r$.
Ak teda vidíme dve rôzne hodnoty, vieme, že tam sú dva kroky rovnakým smerom a teda na súradnici 2 je katakomba, ktorá je hĺbkou na pol ceste medzi tými, na ktoré sme sa opýtali. Odpovieme teda, že na súradnici $x_\star = 2$ má funkcia hodnotu $y_\star = (f(1) + f(3))/2$.

Čo ak $f(1) = f(3)$? V takomto prípade vieme, že na súradnici 2 je lokálny extrém (minimum alebo maximum – nevieme ktoré z toho, ale je nám to jedno). Ako sme si zdôvodnili vyššie, okolo každého extrému je celá funkcia symetrická. Vieme teda, že napr. musí platiť aj $f(0) = f(4)$. Položíme šamanovi otázku s $x = 0$ a potom odpovieme, že na súradnici $x_\star = 4$ má funkcia hodnotu $y_\star = f(0)$.
Túto stratégiu vieme jednoducho zapísať aj v pseudokóde.

```
a = ask(f(1))
b = ask(f(3))
if a != b:
    answer(2, (a+b)/2)
else:
    c = ask(f(0))
    answer(4, c)
```

Rozcvička pred zvyškom riešenia

Dokážeme si, že jedna otázka nestačí.
Majme ľubovoľné riešenie, ktoré sa opýtalo na hodnotu $f(x)$ a potom si skúsilo tipnúť nejakú inú hodnotu $f(z)$. Môže byť takéto riešenie zaručene správne?
Zjavne nie. Existujú totiž aspoň dve možnosti ako katakomby vyzerajú. Všimnime si dve konkrétne.
Jedna možnosť je, že keď ideme od x ku z, tak funkcia f stále rastie o 1, a teda $f(z) = x + |z - x|$.
Druhá možnosť je, že keď ideme od x ku z, tak funkcia f stále klesá o 1, a teda $f(z) = x - |z - x|$.
Keďže tip musí byť iný od otázky, ich vzdialenosť (teda absolútna hodnota rozdielu x a z) je kladná, a teda sme práve popísali dva rôzne systémy katakomb, ktoré sa obe zhodujú na $f(x)$ ale líšia na $f(z)$. Žiaden algoritmus si teda po jednej otázke na hodnotu $f(x)$ nemôže byť istý inou hodnotou $f(z)$.

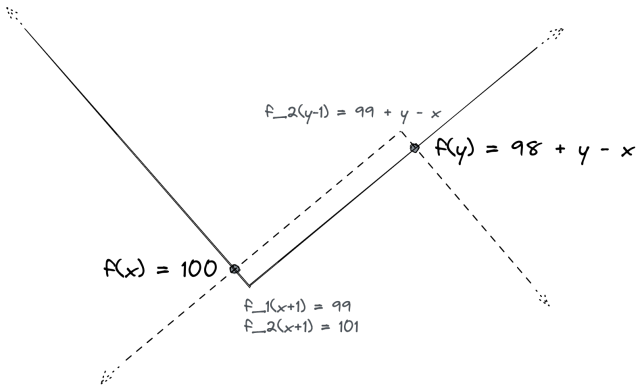
Podúlohy B a C: dôkaz, že sa to lepšie nedá

V tejto časti si rovno ukážeme riešenie ťažšej podúlohy C: dokážeme si teda, že žiadnemu algoritmu nemôžu vždy stačiť len dve otázky, a to ani vtedy, ak navyše zadarmo dostane informáciu, že $r = 1$.
Dôkaz bude myšlienkovo vyzerať podobne ako vyššie uvedená rozcvička. Ukážeme, že nech by sa algoritmus riešiaci našu úlohu prvé dve otázky opýtal akokoľvek, ešte stále budeme pre každú inú súradnicu vedieť nájsť aspoň dva systémy katakomb, ktoré sa zrovna na tej súradnici budú líšiť. Z toho potom vyplynie, že v danej situácii pri žiadnom tipe nemôžeme mať istotu, že odpovieme správne.

Naša konštrukcia bude vyzerať nasledovne:

- Keď sa algoritmus prvýkrát opýta na nejakú pozíciu x, odpovieme mu, že $f(x) = 100$.
- Keď sa algoritmus druhýkrát opýta na nejakú pozíciu y, odpovieme mu, že $f(y) = 98 + y - x$.

Číslo 100 sme si zvolili len tak, hocijaké iné by tiež fungovalo. Číslo $98 - y - x$ sme si zvolili celkom úmyselne, hoci existujú aj nejaké iné, ktoré by fungovali. O chvíľu si dokážeme, že pre tieto hodnoty $f(x)$ a $f(y)$ sa nedá jednoznačne určiť žiadna iná hodnota funkcie $f(z)$ pre ľubovoľné celé číslo z (a to ani ak vieme, že $r = 1$).



Prečo sa nedá sa so znalosťou $f(x) = 100$, $f(y) = 98-y-x$ uhádnuť žiadna konkrétna hodnota $f(z)$? Na obrázku vidíme výsek z dvoch katakomb f_1, f_2, pre ktoré platí, že $f_1(x) = f_2(x) = 100$ a $f_1(y) = f_2(y) = 98 + y - x$. Keď si niekto zvolí súradnicu z, kde by chcel tipovať, my dodefinujeme katakomby z obrázka tak, aby až po z robili to isté, ako na obrázku – tým dosiahneme, že sa v z budú líšiť.
Prvá katakomba (znázornená plnou čiarou) bude definovaná hodnotami $p = 99, q = |x - y| + |x - z| + 5, r = 1, s = x + 1$, čiže nadobúda minimum na pozícii $x + 1$, toto minimum má hodnotu $f_1(x + 1) = 99$, a potom na obe strany stúpa až do vzdialenosti $q = |x - y| + |x - z| + 5$. Táto vzdialenosť bola zvolená tak, aby sa body y aj z nachádzali v intervale $(s - q, s + q)$ aj s rezervou.
Druhú katakombu (znázornenú prerušovanou čiarou) si zvolíme podobne. Maximum bude nadobúdať v bode $s + q = y - 1$ a jeho hodnota bude $f_2(y - 1) = 99 + y - x$. Podobne budeme chcieť, aby bod x aj z sa nachádzal v intervale $(s, s + 2q)$, takže celé parametre sú $p = 99 + y - x - q, q = |y - x| + |y - z| + 5, r = 1, s = y - 1 - q$.
Sami si môžete skúsiť overiť, že v bodoch x a y nadobúdajú obe funkcie naozaj správne hodnoty. Tiež si vieme ľahko spočítať, že

$$f_1(z) = p + |s - z| = 99 + |(x + 1) - z|$$

Podobne

$$f_2(z) = p + q - |(s + q) - z| = 99 + y - x - |(y - 1) - z|$$

Ostáva nám ukázať, že $f_1(z) \neq f_2(z)$. Inak povedané, že $f_2(z) - f_1(z) \neq 0$. Rozoberme si tri prípady, podľa toho, kde sa môže nachádzať z vzhľadom na x a y (o ktorých predpokladáme, že $x < y$).

- z < x

$$f_2(z) - f_1(z) = 99 + y - x - |y - 1 - z| - (99 + |x + 1 - z|)$$
$$f_2(z) - f_1(z) = 99 + y - x - (y - 1 - z) - 99 - (x + 1 - z)$$
$$f_2(z) - f_1(z) = 2(z - x)$$
$$f_2(z) - f_1(z) > 0$$

- x < z < y

$$f_2(z) - f_1(z) = 99 + y - x - |y - 1 - z| - (99 + |x + 1 - z|)$$
$$f_2(z) - f_1(z) = 99 + y - x - (y - 1 - z) - 99 + (x + 1 - z)$$
$$f_2(z) - f_1(z) = 2$$

- y < z

$$f_2(z) - f_1(z) = 99 + y - x - |y - 1 - z| - (99 + |x + 1 - z|)$$
$$f_2(z) - f_1(z) = 99 + y - x + (y - 1 - z) - 99 + (x + 1 - z)$$
$$f_2(z) - f_1(z) = 2(y - z)$$
$$f_2(z) - f_1(z) < 0$$

Vo všetkých prípadoch $f_1(z) \neq f_2(z)$, a to sme chceli dokázať.
Patrilo by sa ešte spomenúť, že je v poriadku používať, z v parametroch f_1 a f_2. My chceme ukázať, že žiadna konkrétna stratégia nefunguje, a takáto stratégia si po vypočutí šamana zvolí konkrétne z, a konkrétne w – tip na hodnotu $f(z)$. Nám v podstate stačí nájsť jednu katakombu, pre ktorú platí $f(z) \neq w$. My sme našli dve z ktorých aspoň pre jednu to platí, čiže tiež dobre. Je tiež v poriadku použiť x, y pri určovaní odpovede na $f(y)$, ale nebolo by v poriadku, ak by sme povedali, že $f(x) = y + z$, lebo ani y ani z ešte v čase odpovedania na prvú otázku nepoznáme.

Iné riešenia a bonus: Existuje aj mnoho iných dôkazov, resp. spôsobov ako môže šaman odpovedať, ktoré by fungovali. Napríklad na voľbe $f(x)$ až tak nezáleží, čiže aj $f(x) = 0, f(y) = y - x - 2$ by fungovalo rovnako.

Iný spôsob je odpovedať, $f(y) = f(x)$, ak ak $x - y$ je párne, alebo $f(y) = f(x) + 1$ ak je nepárne. Môžete sa zamyslieť ako nájsť vhodné dve katakomby pre tento príklad.
Čo by sa ale stalo, ak by sme okrem podmienky $r = 1$ pridali aj ďalšiu podmienku, že $p > 0$? Táto podmienka by spravila podúlohu C o dosť ťažšou, veľa dôkazov zrazu prestane fungovať. Celkom zaujímavé je, že ak zaručene $p > 0$ a dostanem odpoveď, že $f(0) = 100$, $f(1000) = 100$, dokážem jednoznačne určiť, napr. že $f(2023!) = 100$, čiže ak by šaman pre párne $x - y$ odpovedal $f(y) = f(x)$, stačili by nám dve otázky. Môžete sa zamyslieť, prečo je $f(2023!)$ jednoznačne určené. ($2023!$ je 2023 faktoriál, súčin čísel 1 až 2023).
Stále však platí, že sa úloha nedá vyriešiť na menej ako tri otázky. Dokonca stále funguje voľba $f(x) = 100$, $f(y) = 98 + y - x$. Aj keď v našom dôkaze sme využívali, že $p > 0$ pri konštrukcii druhej katakomby (ak by archeológ zvolil dostatočne veľké z, vyšlo by nám p záporné), dá sa zvoliť aj iná katakomba, ktorá by mala kladné p. Skúste ju nájsť. Prvá katakomba je v poriadku, pre ňu $p = 99 > 0$. Celý dôkaz pre $p > 0$ a $r = 1$ si môžete spraviť ako cvičenie.

TRIDSIATY ÔSMY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Jano Hozza, Paulína Smolárová, Timea Szöllősová
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2023