B-I-1 Korytnačie preteky

Táto úloha bude čoskoro použitá aj **v prvom kole aktuálneho ročníka súťaže PRASK**, vzorové riešenie k nej preto zverejníme až po uplynutí termínu na jej odovzdanie. Ak ste však úlohu riešili, budeme radi, ak **zapojíte aj do súťaže PRASK** (https://prask.ksp.sk/), ktorá je od tohto roku určená nielen základoškolákom, ale aj **prvákom stredných škôl**.

B-I-2 Trojuholníky

Prvým krokom je si jasne sformulovať, čo musí platiť pre množinu paličiek, ktoré môžeme vybrať. Nato aby sa z každej trojice paličiek dal vyrobiť trojuholník musí pre každú trojicu paličiek s dĺžkami a, b, c platiť trojuholníková nerovnosť $a+b>c$. Úlohu si tiež môžeme preformulovať tak, že hľadáme najväčšiu podmnožinu paličiek, pre ktorú táto rovnosť nie je porušená.

Priamočiarejšie pomalšie riešenie

Prvým riešením je vyskúšať všetky podmnožiny drievok a pre každú overiť, či neobsahuje trojicu, porušujúcu trojuholníkovú nerovnosť. Podmnožín množiny s veľkosťou n je 2^n. Overenie konkrétnej podmnožiny drievok vieme robiť jednoducho v kubický čase a to tak, že overíme všetky trojice, ktorých je rádovo až $O(n^3)$. Výsledná časová zložitosť tohto priamočiareho riešenia je $O(2^n \cdot n^3)$. Pamäťová zložitosť je iba lineárna - $O(n)$. Môže sa to zdať zlé avšak na nejaký bodový zisk to stačilo.

Zrýchlenie overovania

Nachvíľu sa pozastavme pri druhom kroku nášho prvého riešenia a to overovanie či všetky trojice z podmnožiny spĺňajú trojuholníkovú nerovnosť. To čo vlastne robíme je, že sa snažíme nájsť trojicu, ktorá nerovnosť porušuje. Ako by sme takúto trojicu vedeli nájsť rýchlejšie? Ak sa pozrieme na samotnú nerovnosť $a+b>c$, vieme ju slovami naformulovať takto: Hľadáme jedno číslo (c), ktoré je väčšie ako súčet nejakých dvoch iných(a, b). Ak by sme chceli nájsť trojicu porušujúcu toto pravidlo, je logické, že za c chceme zvoliť maximum z dĺžok drievok, ktoré máme k dispozícii. Súčet a a b chceme zas minimalizovať. Je teda logické, že za tieto hodnoty zvolíme najkratšie drievka, ktoré máme k dispozícii. Rozmyslite si, že ak tieto tri čísla aj tak spĺňajú trojuholníkovú nerovnosť, potom každé 3 čísla z podmnožiny ju spĺňajú tiež. Vidíme, že len pomocou dvoch najmenších a najväčšieho drievka vieme rýchlo rozhodnúť, či všetky trojice spĺňajú trojuholníkovú nerovnosť. Ak si drievka usporiadame, vieme predchádzajúce riešenie zlepšiť a riešiť každú podmnožinu v čase $O(n)$ čo nám zlepší časovú zložitosť nášho riešenia na $O(2^n \cdot n)$.

Vzorové riešenie

Ak by nám niekto povedal, ktoré dve najkratšie paličky a a b budeme mať vo výbere, tak už je tým optimálny výber paličiek jednoznačne určený. Totiž paličky dĺžky $a+b$ a viac zobrať nemôžeme (netvorili by trojuholník s najkratšími dvoma), ani paličky dĺžky menej ako b zobrať nemôžeme (neboli by naše dve paličky najkratšie), no a všetky ostatné paličky môžeme zobrať a dostaneme platné riešenie – a zjavne najväčšie tohto typu.
Na základe tohto pozorovania sa teraz dajú spraviť riešenia súťažnej úlohy v čase $O(n^2)$, prípadne $O(n^2 \log n)$. My však chceme lepšie, tak rozmýšľajme ďalej.

Ak by nám niekto povedal, ktorá palička b bude druhá najkratšia v Timovom výbere, ako zvoliť najkratšiu? Čím bude väčšia, tým väčší bude súčet $a+b$, a teda tým viac dlhých paličiek budeme môcť do výberu pridať. Môžeme teda „pažravo“ za a zobrať paličku bezprostredne predchádzajúcu b.
Teraz teda vieme, že nejaké optimálne riešenie vyzerá tak, že dve najkratšie paličky sú dve po sebe idúce paličky v tomto usporiadanom poradí. No a ak sú tieto dve paličky a a b, tak potom do optimálneho výberu patria aj všetky nasledujúce paličky až po dĺžku $a+b-1$ vrátane.
Môžeme teda postupne vyskúšať všetky možnosti pre najkratšiu paličku vo výbere. Keď vieme najkratšiu (a teda vieme najkratšie dve), vieme použiť binárne vyhľadávanie na to, aby sme našli hranicu medzi paličkami, ktoré sú ešte dostatočne krátke na to, aby patrili do výberu, a paličkami, ktoré už sú pridlhé. Pre každý z n

možných začiatkov takto v čase $O(\log n)$ zistíme, koľko ešte paličiek vieme vybrať. No a tým pádom má aj táto časť nášho riešenia celkovú časovú zložitosť $O(n \log n)$.

Listing programu (Python)

```
N = int( input() )
palicky = [ int(_) for _ in input().split() ]
palicky.sort()

best = min(2, N)

for i in range(N-2):
    male = palicky[i] + palicky[i+1]
    lo, hi = i+1, N
    while hi - lo > 1:
        med = (lo + hi) // 2
        if palicky[med] < male: lo = med
        else: hi = med
    best = max( best, lo-i+1 )

print(best)
```

Ešte lepšie riešenie

Ak by sme paličky dostali na vstupe už usporiadané, išla by táto úloha riešiť aj v lineárnom čase. Predstavme si, že pre nejakú najkratšiu paličku vieme, ktorá najdlhšia palička tiež patrí do výberu. Keď sa teraz posunieme na nasledujúcu paličku a budeme skúšať, čo by sa stalo, keby najkratšou vybratou bola ona, doteraz najdlhšia vybratá palička bude naďalej patriť medzi dostatočne krátke. Novú hranicu treba teda hľadať len od nej ďalej doprava. A to teraz stačí spraviť obyčajným cyklom – postupne sa pozeráme na ďalšie paličky až kým nestretneme pridlhú (alebo koniec poľa).
Dôkaz lineárnej časovej zložitosti tohto riešenia vieme založiť na nasledovnej úvahe: predstavme si, že ľavou rukou ukazujeme na najkratšiu a pravou na najdlhšiu vybranú paličku. Vždy, keď pohnem ľavú ruku o políčko doprava, budem musieť pozerať, či a o koľko pohnem doprava aj pravú. Dôležité ale je, že obe ruky sa hýbu *len smerom doprava*, a teda *dokopy počas celého behu programu* len každou rukou raz prejdem celé pole.

Listing programu (Python)

```
n = int(input())
pal = [int(i) for i in input().split()]
pal.sort()

best, koniec = 0, 0
for i in range(0, n - 2):
    koniec = max(koniec, i + 2)
    prvedva = pal[i] + pal[i + 1]
    while koniec < n and prvedva > pal[koniec]:
        koniec += 1
    best = max(best, koniec - i)

print(best)
```

B-I-3 Delta Nílu

Prirodzeným riešením je postupne simulovať plavbu všetkých k lodí a po jednom prepínať smery v staniciach, ktorými plávajú. Takéto riešenie je však pomalé, lebo pre každú loď budeme musieť prejsť pomerne veľkou časťou delty, navyše po niektorých vodných tokoch budeme prechádzať opakovane.
Chceli by sme preto vedieť pre k-tu loď nájsť rýchlejší spôsob na určovanie smeru, ktorým sa bude plaviť. Potom by nám stačilo iba sledovať jej cestu až do prístavu. Najprv sa zamerajme na úplne prvú stanicu, ktorú pri plavbe stretne každá loď. Nech z tejto stanice vychádza p rôznych tokov, ktoré si očíslujeme od 1 po p, ktoré sa cyklicky striedajú v poradí určenom ich číslami, pričom po toku p nasleduje opäť tok 1.
Keďže cez túto stanicu prejdú všetky lode, pred tým, ako sa tadiaľto plaví loď k nastalo postupne $k-1$ posunutí. Vieme na základe toho určiť, ktorý tok bude nasledovať? Áno, stačí si uvedomiť, že toky sa cyklicky opakujú.

Tokom 1 postupne prejdú lode s číslami $1, p+1, 2\cdot p+1\ldots$ Tokom 2 zase lode $2, p+2, 2\cdot p+2\ldots$ Všeobecne tokom y prejdú lode $y, p+y, 2\cdot p+y\ldots$
Keď teda číslo k zapíšeme do tvaru $x\cdot p+y$, tak hodnota y (musíme ju však zvoliť tak, aby platilo $1\leq y\leq p$) nám určí číslo toku, ktorým sa vydá loď k. No a tento zápis predsa reprezentuje delenie so zvyškom, kde hodnota x je podiel a y je zvyšok, vieme ho preto rýchlo vypočítať.
Teda, je to *takmer* delenie so zvyškom. Pri klasickom delení so zvyškom sú za zvyšok pokladané hodnoty od 0 po $p-1$, čo je od našich hodnôt o 1 posunuté. Aj toto je jedným z dôvodov, prečo informatici uprednostňujú číslovanie, ktoré začína od 0. Pri počítaní s takýmto číslovaním sa totiž pracuje o niečo jednoduchšie ako s tým klasickým od 1. Aby sme sa vyhli zbytočným podmienkam a komplikáciám, radšej si upravíme číslovanie, naše toky teda nebudú očíslované od 1 po p ale od 0 po $p-1$ a aj lode očíslujeme radšej od 0 po $k-1$. Nič sme v skutočnosti nezmenili, ale zrazu už naozaj sedí, že keď hľadáme číslo toku, po ktorom sa bude plaviť posledná loď, teda loď s číslom $k-1$, stačí zobrať jej zvyšok po delení číslom p.
Úspešne sme teda zistili, kam zabočí naša loď pri prvej stanici, vďaka čomu vieme, ktorú stanicu navštívi ako ďalšiu. Vieme však aj pri nej použiť rovnaký trik? Problémom je, že nevieme, koľko lodí k tejto stanici smeruje a teda koľko ich posunulo smer plavby. Práve od tohto počtu posunov totiž závisí výsledné nastavenie. Ako teda túto hodnotu vypočítať?
Vráťme sa ešte na chvíľu k prvej stanici. Vďaka zvyšku po delení sme zistili, ktorým tokom sa musíme vybrať, chceme však zistiť aj to, koľko z tých k lodí tadiaľ išlo. Odpoveďou je, že zhruba každá p-ta loď, keďže sa po p cyklili. Keď sa vrátime k našemu predchádzajúcemu zápisu, hodnotu $k-1$ sme vyjadrili ako $x\cdot p+y$. Hodnota y určovala číslo toku, no a hodnota x určuje práve počet celých cyklov, ktoré na tejto stanici nastali a teda aj počet lodí, ktoré daným tokom smerovali. A nezabudneme započítať aj poslednú loď, teda dokopy sa plaví ďalej tokom y práve $x+1$ lodí.
V tomto momente môžeme využiť to isté riešenie, teda sa tváriť, že ďalšia stanica je v skutočnosti prvá, ktorú tieto lode stretnú. Akurát tých lodí nie je k, ale $x+1$ a nás zaujíma cesta tej poslednej z nich, ktorá má číslo x. Ak má teda ďalšia stanica r tokov, tak opäť vypočítame podiel aj zvyšok po delení hodnoty x číslom r a to nám určí jednak číslo toku, ktorý nasleduje a aj počet lodí, ktoré sa týmto tokom vydali.
Tento postup samozrejme môžeme ďalej opakovať aj pri ďalších staniciach, pričom príslušne redukujeme počet plaviacich sa lodí, až kým nenarazíme na prístav. Časová zložitosť takéhoto riešenia je v podstate nezávislá od hodnoty k, stačí keď prejdeme jednou cestou zadanej štruktúry, ktorej veľkosť je $O(n)$.

Listing programu (Python)

```
n, k = map(int, input().split())
vetvy = [ list(map(int, input().split())) for i in range(n) ]

posledna_lod = k - 1
stanica = 0
while True:
    if vetvy[stanica][0] == -1:
        print(stanica + 1) # toto je pristav
        break
    pocet_tokov = vetvy[stanica][0]
    dalsi_tok = posledna_lod % pocet_tokov
    posledna_lod = posledna_lod / pocet_tokov
    stanica = vetvy[stanica][dalsi_tok + 1]
```

B-I-4 Bitové operácie

Podúloha A (3 body): test či je kladné číslo mocninou dvoch

Kladné číslo je mocninou dvoch, keď má na 1 nastavený práve jeden bit.
V študijnom texte sme v časti „riešenie 2“ videli, že keď zoberieme nejaké nenulové x a vypočítame hodnotu `x and (x-1)`, zmeníme tým poslednú 1 v čísle x na 0.
Vieme teda povedať, že x je mocninou dvoch práve vtedy, ak po tejto operácii už neostanú nastavené na 1 žiadne bity – inými slovami, ak `(x and (x-1)) == 0`.

Podúloha B (3 body): tri jednotky vedľa seba

Tentokrát zafunguje napríklad nasledovný test: `(x and (x shr 1) and (x shr 2)) != 0`.
V tomto teste sa pozeráme na logický and pôvodného čísla x a jeho dvoch kópií, ktoré sú oproti x posunuté o jednu a o dve pozície doprava.
Najľavejšie dva bity výsledku operácie `x and (x shr 1) and (x shr 2)` sú vždy nuly, lebo najľavejšie dva bity `x shr 2` sú nuly. Ďalší bit výsledku je logickým and-om najľavejších troch bitov x, nasledujúci bit je logickým and-om druhého až štvrtého bitu x, a tak ďalej. Ak boli kdekoľvek v x tri jednotky vedľa seba, dostaneme jednotku aj vo výsledku, a ak nie, v každej trojici bitov, ktorej and počítame, bude aspoň jedna nula, a teda vo výsledku budú samé nuly.

Podúloha C (4 body): najvyššia mocnina dvoch deliaca dané kladné číslo

Ľahko nahliadneme, že vlastne máme zistiť, koľkými nulami končí zápis čísla x v dvojkovej sústave. Na určenie tejto hodnoty použijeme binárne vyhľadávanie.
Potrebujeme teda vedieť otestovať, či je na konci x aspoň k núl. Ako to spraviť?
Číslo `(1 shl k) - 1` má hodnotu $2^k - 1$, inými slovami, má k najmenej významných bitov nastavených na 1 a všetky ostatné na 0.
Čo sa stane, keď vypočítame bitový and nášho x a tejto hodnoty? Ak x končí aspoň k nulami, dostaneme zjavne výsledok rovný nule. A naopak, ak je na konci x núl menej ako k, tak poslednú jednotku v x budeme andovať s jednou z jednotiek v `(1 shl k) - 1`, a teda dostaneme nenulový výsledok.
Nižšie uvádzame jednu možnú implementáciu tohto riešenia v Pythone. Operátor `&` v Pythone je bitový and, operátor `<<` je posun doľava.

Listing programu (Python)

```
def konci_aspon_k_nulami(x, k):
    return (x & ((1 << k) - 1) == 0)

def zisti_pocet_nul(x):
    # invariant: x končí aspoň lo, ale ostro menej ako hi nulami
    lo, hi = 0, 64
    while hi - lo > 1:
        med = (lo + hi) // 2
        if konci_aspon_k_nulami(x, med):
            lo = med
        else:
            hi = med
    return lo
```

Časová zložitosť tohto riešenia je logaritmická od veľkosti registra, teda $O(\log B)$. Ak napríklad máme 64-bitové celočíselné registre, tak na zistenie počtu núl na konci x stačí spraviť len šesť testov.

Existuje aj veľa iných spôsobov ako robiť kontrolu počas binárneho vyhľadávania. Napríklad by sme mohli x deliť so zvyškom príslušnou mocninou dvoch, alebo by sme mohli x posunúť o k pozícii doprava a zase doľava a pozrieť sa, či sa tým zmenilo.

Poznámka k hodnoteniu: Zadanie úlohy bohužiaľ nebolo formulované dostatočne jednoznačne, a tak niektorí riešitelia riešili ľahšiu úlohu: namiesto nami zamýšľaného zistenia exponentu len zistili samotnú hodnotu tejto mocniny (teda napríklad pre $x = 24$ nevypočítali hodnotu 3 ale hodnotu $2^3 = 8$).
Túto verziu úlohy vieme riešiť v konštantnom čase: už vieme, že v `x and (x-1)` práve ten hľadaný bit čísla x chýba, takže v `x xor (x and (x-1))` je len ten hľadaný bit nastavený.
Keďže nejasná formulácia zadania je našou chybou, aj riešenia tejto verzie úlohy sme uznávali ako správne.

TRIDSIATY ŠIESTY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek, Andrej Korman
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2020