Vzorové řešení soutěže Kasiopea

1 Potřebný výpočet

Hlavním cílem této úlohy bylo naučit se efektivně mocnit čísla. Ukážeme si, jak se úloha dá vyřešit v čase $O(\log a + \log b)$.

Nejdříve si předpočítáme mocniny $a^1, a^2, a^4, a^8, \ldots$ Poté využijeme toho, že $a^x \cdot a^y = a^{x+y}$ a podle binárního zápisu čísla b vynásobíme příslušné předpočítané mocniny a a dostaneme a^b. Obdobně spočítáme i b^a.

Při samotném programování dokážeme mocninu spočítat dokonce i bez předvýpočtu. Zároveň si budeme pamatovat mezivýsledek, aktuální počítanou mocninu a jen budeme zjišťovat, zda je na daném místě v binárním zápise exponentu jednička. Viz zdrojový kód programu.

Při samotné implementaci si musíme dát pozor, abychom při každém výpočtu modulili a používali datový typ `long long`, jinak bychom brzy přetekli. A konečně si musíme dát pozor na modulení záporných čísel, které ve většině jazycích vychází záporně.

2 Koně na šachovnici

Máme pět koní a pět cílových políček. Hodilo by se nám zjistit, jak je každý kůn od každého cílového políčka daleko. K tomu použijeme algoritmus prohledávání do šířky a pro každého koně zvlášť zjistíme, jaký je nejmenší počet skočení, kterými se může dostat do jednotlivých cílových políček.

Nyní se už jen podíváme na všechny možnosti toho, kam který kůň může doskákat, a z nich vybereme tu nejlepší. Možností je jen $5! = 120$, takže je můžeme vyzkoušet všechny. Pozor existuje příklad vstupu, kdy koně není možné do cílové pozice přeskládat. Schválně zkuste přijít na to, jaký to je.

Při implementaci je důležité nespouštět prohledávání do šířky při každé ze 120 možností, ale jen jednou na začátku a výsledky si zapamatovat. Dále na konci nesmíme zapomenout ověřit, že jsme nějaké řešení opravdu našli. To se obvykle dělá tak, že si nejlepší řešení zinicializujeme na `INT_MAX`, nebo jinou velkou hodnotu a před výpisem výsledku ověříme, zda se hodnota změnila.

Řešení má časovou složitost $O(N^2+5!) = O(N^2)$, kde N je rozměr šachovnice. Paměťová složitost je také $O(N^2)$.

3 Kufr s XORem

Tato úloha je modifikací na známou úlohu jménem "Problém batohu". Rozdíl je v tom, že kromě maximalizace objemu musíme navíc zajistit maximalizaci XORu a počtu věcí. Celkově však budeme postupovat velmi podobně.

Vytvoříme si tabulku velkou $(M+1)\times(M+1)$, kde první rozměr bude udávat zaplněnost batohu a druhý rozměr XOR věcí. Do téhto tabulky si postupně budeme psát, kolik věcí do kufru dokážeme dát, aby splňovaly parametry daného políčka. Na začátku v celé tabulce budou -1, akorát na souřadnicích $(0,0)$ bude 0, tj. stav se kterým začínáme.

Nyní budeme tabulku postupně vylepšovat. Z tabulky pro žádnou věc získáme tabulku pro jednu věc, z ní pak tabulku pro dvě věci, ... Pokud máme tabulku výsledků pro $i-1$ věcí, tak z ní vyrobíme tabulku pro i věcí následujícím způsobem:

1. Pro všechny váhy m od M do vaha$[i]$, pro všechny x od M do 0 dělej:
2. Pokud existuje řešení na souřadnicích $[m - \text{vaha}[i]; x \oplus \text{vaha}[i]]$ a je možné pomocí něj vylepšit řešení na $[m, x]$, tak jej vylepši.

Po úpravě tabulky s $(i-1)$ věcmi dostaneme tabulku nejlepších výsledků pro i věcí, protože jsme jen zkusili zlepšit všechny nejlepší výsledky s $(i-1)$ věcmi. Tím, že jsme tabulku zlepšovali od největších vah máme zajištěné, že vylepšované údaje pochází ještě z předchozí iterace tabulky. Tato technika je v informatice často používána a patří do kategorie, která se nazývá Dynamické programování.

Časová složitost algoritmu je $O(M^2 \cdot N)$ a paměťová složitost je $O(M^2)$. Na závěr ještě poznamenejme, že XOR věcí nikdy nemůže být větší než jejich celková váha. Zkuste si schválně ověřit, že je to pravda.

4 Cesta přes celnice

Kdyby v úloze žádné celnice nebyly, tak bychom úlohu vyřešili klasickým Dijkstrovým algoritmem. Celnice tam ale jsou a tento postup nám kazí. Co s tím?

Přesto použijeme Dijkstrův algoritmus, akorát namísto vzdáleností, které jsme ujeli, budeme porovnávat dvojice čísel, skládající se z počtu celnic, přes které jsme projeli, a počtu kilometrů, které jsme ujeli. Počet celnic při cestě z místa A do místa B budeme zvyšovat právě tehdy, když místo B je celnicí a není cílovým místem.

A to je vlastně celé. Časová složitost je $O((N+M)\cdot \log M)$. K získání plného počtu bodů bylo nutné pro počet kilomentrů používat typ `long long`.

5 Divný jazyk

Tato úloha byla ze všech asi nejtrikovější. Jak slova rychle rozdělit do skupinek nezávisle na rotaci?

Nejdříve se nám bude hodit si všechna slova převést do nějakého standardizovaného tvaru, ve kterém všechna slova patřící do stejné skupinky budou

vypadat stejně. Tento požadavek splňuje například lexikograficky nejmenší rotace slova. Stačí nám tedy u každého slova najít jeho lexikograficky nejmenší rotaci a vyrobit `trii`, ve které si každý vrchol pamatuje, kolik slov v něm končí.

Jediný problém na tomto algoritmu je najít rychle onu lexikograficky nejmenší rotaci. Lexikograficky nejmenší rotace určitě bude začínat nejnižším znakem ve slově. Vezmeme tedy všechny tyto znaky jako možné začátky a podíváme se na jejich druhou pozici. Z nich uchováme jen ty, které jsou nejnižší mezi druhými pozicemi. Ve zbytku se stejným způsobem podíváme na znaky na třetích pozicích atd.

V některém kroku se nám však může stát, že na nějakých následujících pozicích jsou opět nejmenší znaky slova. To jsou naše původní začátky, u kterých již máme spočítáno jak dlouho byly ve výběru nejmenších možností. Pomocí těchto informací tedy prodloužíme jen ty možnosti, které pokračují nejdelšími již spočítanými částmi, a ostatní zahodíme. Dále se posuneme o délku těchto částí a pokračujeme na další pozici. Nakonec nám buď zbyde jen jedna možnost, a nebo se všechny možnosti na sebe napojí. V případě napojení bez újmu na obecnosti vezmeme tu první z nich. Tuto pozici pak prohlásíme za začátek lexikograficky nejmenší rotace. Pokud během algoritmu za sebe napojujeme několik stejně velkých úseků, tak si nadále ponecháme jen tu první z nich, sami si rozmyslete proč.

Časová složitost nalezení lexikograficky nejmenší rotace je $O(d)$, kde d je délka slova. Na každý znak se vlastně koukneme jen dvakrát, výjimečně třikrát, a jen konstantakrát napojujeme úseky. Časová složitost celého algoritmu je tedy $O(D)$, kde D je celková délka všech slov, protože s `trií` umíme pracovat lineárně vzhledem k délce slov. Paměťová složitost je lineární, ale pozor: `trie` má v sobě poměrně velkou multiplikativní konstantu, proto je potřeba pamětí šetřit.

Jiným možným řešením bylo použít hashování namísto `trie`. Hashování sice obecně nefunguje bez ověřování, ale pravděpodobnost nefunkčnosti je tak malá, že se při soutěžích standardně nebere v potaz.

Karel Tesař