Korespondenční Seminář z Programování
ZAČÁTEČNICKÁ KATEGORIE
37. ročník **KSP-Z** **Prosinec 2024**

Řešení druhé série začátečnické kategorie 37. ročníku KSP

37-Z2-1 Tiskárna na kečup

Nejprve si musíme uvědomit několik věcí. Zuzka ví, kolik má kečupu v litrech, ale zákazníci vědí, kolik mililitrů kečupu potřebují na svůj toast. Protože se v počítači celá čísla lépe reprezentují než čísla desetinná, vynásobíme počet litrů kečupu K tisícem a získáme počet mililitrů.

Teď, když jsme provedli konverzi, můžeme načítat toasty ze vstupu. Pro každý toast zkontrolujeme, zda máme dostatek kečupu. Pokud ano, vypíšeme číslo tohoto toastu a odečteme spotřebovaný kečup od celkového množství K. Pokud ne, toast přeskočíme. Pozor, můžeme vytisknout i ten toast, který má přesně tolik kečupu, kolik máme.

Když projdeme všechny toasty, jsme hotovi. Všechny vstupy se vešly do 32-bitových hodnot, tedy v `C++` stačilo použít `int`. V Pythonu toto dokonce není potřeba řešit vůbec.

Složitost algoritmu

Podívejme se na složitost algoritmu. Každou objednávku na toast dokážeme zpracovat v konstantním čase, protože uděláme jen jedno porovnání a odečtení. Jelikož je toustů T, zpracování všech bude trvat čas $\mathcal{O}(T)$

Co se týče paměťové složitosti, můžeme využít toho, že nám stačí pamatovat si pouze zbývající počet mililitrů kečupu a informace o aktuálně zpracovávaném toastu, tedy jeho cenu a pořadí. V případě, že se rozhodneme toast vydat, můžeme ihned vypsat na výstup jeho pořadové číslo a začít zpracovávat další toast. Celková paměťová složitost je tedy $\mathcal{O}(1)$

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/37-Z2-1.py`

Úlohu připravili: Vojta Káně, Matúš Púll, Vladimír Sklenár

37-Z2-2 Sekání obilí

Na vstupu jsme dostali posloupnost časů, kdy bylo obilí na jednotlivých pozicích zasazeno. My ale potřebujeme vědět, jak moc ho na pozicích je. Pokud je aktuální čas T a obilí na daném políčku bylo zasazeno v čase P_i, pak je nyní $T - P_i$ kostiček vysoké. Tím máme způsob, jak zjistit, kolik obilí na konkrétní pozici může Kevin sklidit.

Teď potřebujeme zjistit, kde se nejvíc vyplatí posekat M sousedních pozic. Na to nám stačí projít celé pole a zjistit, kolik obilí sklidíme na každé M-tici sousedních polí. Toho můžeme docílit tak, že si pamatujeme součet aktuálních M polí, na které se zrovna díváme, a postupně posouvat toto okénko o 1 dál, dokud se nedostaneme na konec. Posunutí okénka o jedno pole doprava docílíme tak, že nejlevější políčko odebereme a přidáme políčko následující.

Náš algoritmus bude vypadat tak, že na začátku sečteme všechny výšky obilí na prvních M místech. Při posunutí okénka o jedno políčko doprava od tohoto součtu odečteme výšku obilí nejlevějšího políčka okénka a přičteme výšku obilí následujícího políčka. Po každém kroku se také podíváme, zda aktuální součet není dosavadní maximum. Tedy si pamatujeme v průběhu nejvyšší množství posekaného obilí a kde jsme ho posekali. Ve chvíli, když jsme na úseku, na kterém bychom posekali víc obilí, upravíme své dosavadní maximum na tento úsek. Po projití okénkém celého obilného pole jsme zkontrolovali všechny souvislé úseky délky M a můžeme si tak být jistí, že jsme nalezli ten, jehož posečením dostaneme nejvíce kostiček obilí.

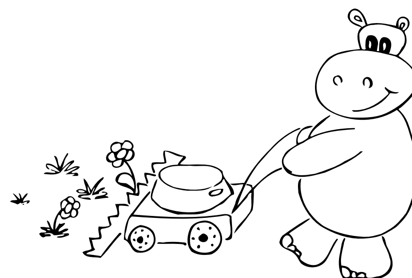
Složitost algoritmu

Teď pojďme rozebrat časovou složitost algoritmu. Dokážeme v konstantním čase vyhodnotit, kolik obilí na pozici vyrostlo tím, že odečteme čas zasazení od aktuálního času. Následně okénkem procházíme pole všech časů zasazení obilí, přičemž na každé políčko se podíváme nejvýše dvakrát – jednou ho do okénka přidáme a jednou odebereme (až na posledních M prvků, které nemusíme odečítat). Na každém z P políček tak strávíme konstantní množství času, což dává složitost v $\mathcal{O}(P)$.

V paměti si držíme pole, které má P prvků. Také si průběžně pamatujeme součet aktuálních M pozic a informace o úseku s doposud maximálním součtem výšek obilí. To nám dává paměťovou složitost $\mathcal{O}(P)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/37-Z2-2.py`

Úlohu připravili: Kačka Doubková, Matúš Púll



37-Z2-3 Hodní orgové

Jen tak na úvod

Hned na úvod se hodí říct, kde byl zakopaný pes téhle úlohy a jak se zdánlivě těžká úloha dostala do kategorie Z. V zadání záměrně chyběla dost podstatná informace, která ulehčuje a možná i prozrazuje řešení, že úloh je maximálně 7. Toto se ale dalo zjistit vygenerováním posledního nejtěžšího vstupu, kde bylo vidět daných 7 úloh. Díky tomuto víme, že všech možných permutací bodů úloh je "jen" 7!, což je v pohodě upočítatelné hrubou silou.

Co je to vlastně ta permutace?

Představte si, že máme pole čísel $[1, 2, 3, 4]$. Permutace je uspořádání ("prohození") těchto prvků ("čísel"), kde se každý prvek objevuje právě jednou. Takové permutace můžou být například $[2, 3, 1, 4]$ nebo $[4, 1, 2, 3]$. Důležité je, že počet všech permutací je $n!$, kde n je počet prvků. Pokud

bychom dokázali vymyslet způsob, jak všechny permutace najít, mohli bychom si vyzkoušet všechna možná obodování úloh v sérii a jednoduše potom ověříme, kolik účastníků bude úspěšných. Jen pro úplnost, v naší úloze se můžeme setkat s tzv. permutací s opakováním, což znamená, že se prvky permutace mohou opakovat (např. $[1, 2, 2, 4]$). Celkový počet permutací s opakováním je menší než počet permutací, nám to ale nevadí.

Algoritmus

Pojďme si popsat algoritmus. Vzhledem k tomu, že úloh je málo, vygenerujeme všechny možné permutace. Pro každou jednu permutaci spočítáme počet úspěšných řešitelů a po cestě si budeme ukládat zatím nejvyšší počet úspěšných řešitelů a společně s tím i nejlepší obodování úloh. Tedy, pro každou permutaci projdeme všechny řešitele, podíváme se, které úlohy daný řešitel může splnit a podle toho řešiteli přičteme body. Pokud má bodů více, než je limit úspěchu, pak je řešitel úspěšný. Až výpočet doběhne, budeme mít nejlepší obodování úloh a počet úspěšných řešitelů. Zde je nutno dodat, že řešení se v závislosti na pořadí generování permutací bodů za úlohy může lišit, pro jedno zadání tedy může být více správných řešení.

Jak generovat permutace?

Existuje více typů algoritmů, jak generovat permutace, každý s trochu jiným přístupem, kombinující různé výhody. My si popíšeme jeden z jednodušších, rekurzivní algoritmus s prohazováním prvků na místě. Algoritmus dostane na vstupu pole čísel a "začátek" - pozici, kde budeme pracovat s aktuálním prvkem. Pokud je tato pozice rovna délce vstupního pole, znamená to, že jsme prošli všechny prvky, a pole v jeho současném stavu představuje jednu z permutací. Tuto permutaci následně použijeme na výpočet počtu úspěšných řešitelů. Jinak iterujeme přes všechny prvky od aktuálního prvku až do konce pole. Každý prvek na dané pozici (dočasně) prohodíme s aktuálním prvkem a poté rekurzivně zavoláme algoritmus, přičemž posuneme „začátek" o jednu pozici doprava, abychom zpracovali jen zbytek pole. Jakmile se rekurzivní volání vrátí, prohodíme prvky zpět, abychom pole obnovili do původního stavu. Tento proces postupně vygeneruje všechny možné uspořádání prvků v poli, protože každé prohození a následná rekurze reprezentují jinou cestu, jak uspořádat prvky. Ve vzorovém řešení používáme pythonní knihovnu itertools, která má přímo funkci na generování permutací.

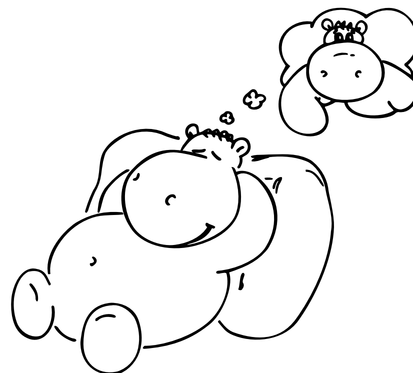
Časová a paměťová složitost

Časová složitost generování všech permutací je $\mathcal{O}(U \cdot U!)$, kde U je počet úloh, protože počet všech možných permutací je $U!$ a pro vygenerování jedné děláme $\mathcal{O}(U)$ kroků. Paměťová složitost je lineární vzhledem k velikosti vstupu, tedy $\mathcal{O}(U)$, algoritmus na generování permutací vše dělá v jednom poli a žádné další nevytváří.

Celková časová složitost je $\mathcal{O}(U \cdot U! \cdot R \cdot U)$, kde U je počet úloh a R je počet řešitelů, protože musíme vygenerovat všechny permutace a pro každou vygenerovanou permutaci pak musíme projít všechny řešitele a práce na spočítání bodů jednoho řešitele je $\mathcal{O}(U)$. Celková paměťová složitost je $\mathcal{O}(U)$, protože navíc k paměti pro generování permutací si pamatuje jen maximální počet řešitelů a jedno nejlepší obodování.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/37-Z2-3.py`

Úlohu připravili: Petr Budai, Michal Kodad, Vladimír Sklenár



37-Z2-4 Udatná želva

Uvažujme nejprve, že by se výbuch šířil jen ve vodorovném směru. Pak bychom mohli jednoduše pro každé pole spočítat na kolik polí se výbuch rozšíří. Budeme procházet tabulku po řádcích a vždy, když narazíme na nezničitelný kámen, nebo začátek či konec řádku, si zapamatujeme, kdy tento úsek začíná. Pak si až do dalšího nezničitelného kamene, nebo konce řádku budeme pamatovat, kolik jsme potkali zničitelných kamenů. Až narazíme na konec našeho úseku, tak prostě skočíme na začátek úseku a projdeme všechny jeho pole ještě jednou a do každého z nich si uložíme, kolik kamenů v úseku je. Tolik jich výbuch tímto směrem zničí.

Toto můžeme následně udělat i procházením tabulky po sloupcích, akorát budeme počty kamenů přidávat k těm, co už v polích jsou. Všimněme si, že nyní je v každém políčku zapsáno, kolik kamenů zničí dohromady v horizontálním a vertikálním směru. Stačí tedy najít políčko s největším číslem a máme náš výsledek. A na to nám stačí tabulku jen celou projít.

Ještě je potřeba podchytit jeden drobný detail (právě v těch se ale často skrývá ďábel). Pokud bychom byli na poli se zničitelným kamenem, tak jsme ho aktuálně započítali dvakrát. Jednou horizontálně a jednou vertikálně. V takovém případě nám stačí jedničku zase odečíst a vše bude vycházet.

Algoritmus navštíví každé políčko nejvýše 5krát – dvakrát při vodorovném i svislém směru a jednou při závěrečném hledání nejlepšího políčka. Běží tedy v čase $\mathcal{O}(N)$, kde N je počet políček v jeskyni. Potřebuje také $\mathcal{O}(N)$ paměti, protože pro každé políčko si potřebuje pamatovat jedno číslo.

Úlohu připravil: Adam Jahoda

matfyz

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.
Realizace projektu byla podpořena Ministerstvem školství, mládeže a tělovýchovy.

Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Organizátoři a kontakty: *https://ksp.mff.cuni.cz/kontakty/*