Korespondenční Seminář z Programování

ZAČÁTEČNICKÁ KATEGORIE

32. ročník | **KSP-Z** | **Říjen 2019**

Řešení první série začátečnické kategorie 32. ročníku KSP

32-Z1-1 Kevin v papírnictví

K tomu, abychom za věci utratili co možná nejméně, potřebujeme do košíku postupně vybírat věci od té nejlevnější po tu nejdražší.

Pole cen tedy vzestupně setřídíme a budeme jej postupně procházet, přičemž si budeme pamatovat součet cen věcí, které při průchodu potkáme. Jakmile při procházení přesáhne tento součet koruny vyhrazené na nákup, tak podle pozice v poli víme, kolik věcí si můžeme koupit.

Jelikož se však zadání ptá na počet věcí, které si koupit nemůžeme, tak jako řešení vypíšeme počet věcí od pozice, kde jsme skončili s procházením, do konce pole cen.

Řešení je jistě správné, určitě se totiž nemůže stát, že bychom si mohli koupit víc věcí, než nám vypíše algoritmus. Vyhodili jsme z košíku ty nejdražší věci, které jsme mohli, takže nám jich určitě nestačilo vyhodit méně.

Rychlost řešení ovlivní hlavně to, jak rychle dokážeme ceny setřídit. Existují různé třídící algoritmy a pokud použijete nějaký algoritmus zabudovaný přímo v programovacím jazyce, běží většinou v čase $\mathcal{O}(n \log n)$. Pokud jste si implementovali vlastní jednoduché třídění, pravděpodobně bude mít časovou složitost $\mathcal{O}(n^2)$. O třídících algoritmech si můžete přečíst v naší kuchařce o třídění.[1]

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z1-1.py`

Program (C++):
`http://ksp.mff.cuni.cz/viz/32-Z1-1.cpp`

Tom Sláma

32-Z1-2 Chybná účtenka

V úloze máme pro každou položku na účtence, což je posloupnost znaků, určit, zda obsahuje nějakou jinou posloupnost znaků, v tomto případě název položky, kterou si Kevin opravdu koupil, jako vybranou podposloupnost – to je taková posloupnost, která z původní vznikne vyškrtáním některých prvků. Této hledané posloupnosti budeme říkat *jehla* (což je termín vypůjčený z popisu algoritmů, které se zabývají vyhledáváním souvislého podřetězce v textu, neboli hledáním jehly v kupce sena).

Protože jsou jednotlivé řádky vstupu představující položky na účtence na sobě nezávislé, budeme se zabývat řešením pouze jednoho z nich. Výsledný algoritmus poté jednoduše spustíme na každý řádek zvlášť.

Všimněme si, že pokud řádek obsahuje jehlu, tak její písmena se budou na řádku vyskytovat ve stejném pořadí, jako v jakém jsou v jehle.

Postupně projdeme všechna písmenka v řádku zleva doprava. Nejdříve hledáme první písmeno jehly. Dokud se písmena na řádku nerovnají tomuto písmenu jehly, tak je pomyslně škrtáme. Až nalezneme první písmeno jehly, tak na zbývajících písmenech řádku opakujeme stejný postup, ovšem nyní hledáme druhé písmeno jehly, poté třetí a tak dále.

Implementovat tento postup můžeme například tak, že postupně projdeme všechna písmena řádku, a navíc si pamatujeme index písmena jehly, který hledáme, a ten zvýšíme pokaždé, když se písmeno na řádku rovná hledanému písmenu. Jakmile nalezneme poslední písmeno jehly, víme, že se jehla na řádku vyskytuje. Pokud je ale po průchodu řádku index hledaného písmena menší než délka jehly, jehla se na řádku nevyskytuje.

Tímto algoritmem nalezneme všechna písmena jehly právě tehdy, když řádek obsahuje jehlu jako vybranou podposloupnost. Proč je tomu tak?

Je snadné vidět, že pokud takto najdeme všechna písmena jehly, tak řádek jehlu obsahuje. Těžší je ukázat, že se nemůže stát, že by v řádku jehla byla, ale algoritmus ji nenašel. Uvažujme nějaký řádek, ve kterém se jehla vyskytuje. Pokud by ji algoritmus nenašel, tak musel při hledání nějaké i-té písmeno jehly přeskočit, což je může stát jen tehdy, pokud v úseku řádku před tímto písmenem nenašel ani předchozí $i-1$ písmeno jehly, což se stane jen když nenajde před $i-1$ ani $i-2$ písmeno a tak dále. Víme ale, že se v řádku první písmeno vyskytuje na nějaké pozici j, a to algoritmus určitě najde, i když možná na pozici menší než j, což nám ovšem nevadí. Druhé písmeno jehly se vyskytuje až za pozicí j, tudíž jej algoritmus taky najde. Podobně najde všechna písmena jehly. Algoritmus tedy jistě odpoví správně.

Jeho časová složitost je lineární vzhledem k délce řádku – na každé písmeno se podívá nejvýše jednou a pro každé písmeno se v jehle posuneme také nejvýše jednou.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z1-2.py`

Program (C++):
`http://ksp.mff.cuni.cz/viz/32-Z1-2.cpp`

Kuba Pelc

32-Z1-3 Školní knihy

Všimněme si, že otáčení libovolné knihy ovlivňuje pouze, jaký úhel svírá s knihou pod ní. Otočením jedné knihy totiž zároveň otočíme i všechny knihy nad ní, úhly mezi nimi se tedy nezmění. Proto nám stačí spočítat, o kolik musíme pootočit každé dvě sousední knihy, aby svíraly stejný úhel.

[1] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

Hledáme tedy pro každou dvojici knih takový úhel, o který musíme vrchní knihu otočit, abychom ji vyrovnali s tu spodní.

Otáčet každou knihu budeme podle rozdílu úhlů s knihou pod ní. Jelikož však v kontextu úlohy dává smysl otáčet pouze o kladné úhly, budeme počítat s absolutními hodnotami těchto rozdílů.

Problém je, že takovéto úhly nemusí být minimální – rozdíl může být větší než 180 stupňů (viz. ukázkový vstup) a otáčeli bychom tak o více, než je nutné. V takovém případě ale stačí těchto 180 stupňů odečíst a otáčet tak „na druhou stranu".

Našli jsme tedy pro každou knihu nejmenší úhel, o který je nutné ji otočit. Tyto úhly můžeme postupně přičítat k nějaké proměnné a nakonec toto číslo vypsat.

Pro každou knihu provedeme pouze konstantně mnoho operací, proto bude časová složitost $\mathcal{O}(n)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z1-3.py`

Program (C++):
`http://ksp.mff.cuni.cz/viz/32-Z1-3.cpp`

Tom Sláma

32-Z1-4 Plánek školy

Měli jste za úkol v zadaném poli najít co největší prostor s volnými políčky. Nejprve je potřeba načíst vstup do paměti. Pak ho začneme postupně (třeba po řádcích) procházet. Dokud potkáváme jen zdi, pokračujeme dál. Jakmile však narazíme na místnost, chtěli bychom ji nyní celou prozkoumat, než budeme pokračovat v postupném procházení políček.

Použijeme algoritmus prohledávání do šířky. Vytvoříme si frontu, do které budeme přidávat políčka, která musíme prohlédnout. Nejprve do ní přidáme naše první políčko místnosti, na které jsme vstoupili. Při přidávání do fronty zároveň označíme toto políčko, že jsme na něm už byli, a nebudeme ho tedy navštěvovat znovu. K tomu si můžeme do reprezentace plánku školy přidat třetí symbol: k tečce `.` jako prázdné místnosti a křížku `X` jako stěně můžeme přidat třeba `B` pro místnost, ve které jsme už byli.

Poté vždy vezmeme první prvek fronty, přičteme jedničku k velikosti aktuálně procházené místnosti, a podíváme se na všechny 4 sousedy tohoto políčka. Pokud je některý z nich místnost a ještě jsme v ní nebyli, označíme si ji jako projitou a přidáme ji do fronty.

Když bude fronta prázdná, prošli jsme celou místnost. Určitě jsme na žádné políčko patřící k místnosti nemohli zapomenout, protože musí sousedit s nějakým dalším políčkem této místnosti a při jeho procházení jsme do fronty museli přidat i toho políčko.

Jakmile projdeme celou místnost, pokračujeme v postupném procházení políček od místa, kde jsme skončili, a hledáme další prázdnou místnost. Tu, kterou jsme právě prošli, již znovu procházet nebudeme, její velikost jsme již spočítali a máme všechna její políčka správně označená, abychom poznali, že jsme tuto místnost již navštívili.

Jaká bude časová složitost algoritmu? Každé políčko projdeme právě jednou při postupném procházení. Dále pak na některá políčka spouštíme prohledávání do šířky. Tím ale s každým políčkem také strávíme pouze konstantně mnoho času. Nejvýše jednou ho přidáme do fronty a nejvýše jednou se na něj podíváme z každého souseda. Proto bude časová složitost algoritmu $\mathcal{O}(n)$, kde n je celková počet políček v plánku.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z1-4.py`

Tom Dostál

32-Z1-5 Nábor basketbalistů

Naším úkolem bylo vybrat dvanáct nejvyšších hráčů. Aby se nám lépe počítalo, řekněme že hráčů na výběrové řízení přišlo N, a vybíráme $K = 12$ nejvyšších. Také se domluvme, že výšky hráčů jsou nějaká desetinná čísla s neznámou přesností, ať to nemáme tak jednoduché. Také nechť žádní dva hráči nejsou stejně vysocí.

Přímočarým řešením může být všechny hráče seřadit od nejvyššího, a vzít jich prvních dvanáct. To ale vyžaduje znát nějaký algoritmus na třídění, který poběží nejlépe v $\mathcal{O}(N \log N)$. Mohli byste namítnout, že třídit čísla umíme i v lineárním čase – to ovšem umíme pouze tehdy, jsou-li čísla celá, nebo mají alespoň omezenou přesnost.

Pojďme to zkusit nejprve nějakým hloupým třídícím algoritmem, například `SelectSort`em. Ten pracuje tak, že vybere ze všech čísel maximum, to vypíše na výstup a odebere ho ze vstupu. Toto opakuje, dokud na vstupu zbývají čísla. Jak toto realizovat v praxi? Nejprve načteme všechna čísla do seznamu. Z něj pak odebereme nějaké číslo (třeba poslední, to se odebírá snadno) jako kandidáta. Poté projdeme celý vstup a každý prvek porovnáme s kandidátem. Pokud narazíme na číslo větší, tak jej s kandidátem vyměníme. Na konci nám zbude jako kandidát číslo, které je větší nebo rovno všem ostatním – hledané maximum. Jako bonus jsme ho rovnou ze seznamu odebrali, byť jsme při tom seznam drobet zamíchali. To ale ničemu nevadí. Postup opakujeme, dokud zbývají čísla v seznamu.

Jak je tento algoritmus rychlý? Pro každý prvek na vstupu projde až celý vstup – tedy $\mathcal{O}(N^2)$. Počkat, my jsme ale nepotřebovali seřadit všechna čísla, nám přeci stačí prvních dvanáct! Pokud se po vypsání dvanáctého čísla zastavíme, projdeme celý vstup pouze dvanáctkrát, což dává časovou složitost $\mathcal{O}(N)$. Lepší už to být nemůže, neboť se vždy musíme na každé číslo alespoň jednou podívat, jinak by se nám pod něj mohlo schovat třeba zrovna to nejvyšší.

Co kdyby ale K nebylo zrovna dvanáct? Pak by náš algoritmus měl složitost $\mathcal{O}(NK)$, a byl by zde prostor pro zlepšení. Tak pojďme na to.

Místo jednoho kandidáta si jich vezmeme rovnou K. Pak pro každý další prvek x ze vstupu najdeme toho nejmenšího z kandidátů m, kterého porovnáme s x. Pokud je x větší, nahradíme s ním m. Na konci nám zbude hledaná podmnožina K nejvyšších prvků. Pomohli jsme si? Nikoliv, složitost je stále $\mathcal{O}(NK)$. Algoritmus v tuto chvíli zdržuje nalezení nejmenšího kandidáta. Zrychlíme ho s použitím datové struktury jménem halda.[2] Stačí nám úplně obyčejná halda, žádné pokročilé vlastnosti nevyužijeme. Haldu vytvoříme nad kandidáty, díky čemuž zrychlíme krok algoritmu (nalezení minima a možnou výměnu za nový prvek) na $\mathcal{O}(\log K)$, tedy dohromady celý algoritmus na $\mathcal{O}(N \log K)$.

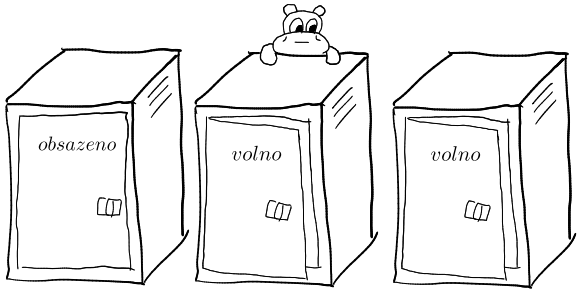
[2] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`

Na závěr dodejme, že existuje i řešení, které je v nejhorším případě lineární s N. Idea je taková, že stačí nalézt K-tý největší prvek a nakonec vypsat všechny prvky, které jsou větší. Algoritmus na hledání K-tého nejmenšího prvku pracuje následovně: rozdělí prvky do pětic, v pěticích najde v konstantním čase mediány, poté najde (rekurzivně) medián mediánů pětic P. Nakonec spočítá pozici P v původním seznamu a rozdělí prvky na menší a větší než P. Pokud není samo P hledaným prvkem, zavolá sám sebe znovu na jednu ze skupin pro upravené K. Až důkladnou analýzou tohoto algoritmu zjistíme, že běží v čase $\mathcal{O}(N)$, taková analýza je ale dalece nad rámec začátečnické kategorie. Navíc, v časové složitosti algoritmu vycházejí vysoké konstanty, takže se v praxi nepoužívá.

Ondra Hlavatý

32-Z1-6 TOI TOI

Zkusme úlohu vyřešit hezky jednoduše. Mějme pole velikosti N, ve kterém si u každé kabinky budeme pamatovat, jestli je obsazená nebo není – `volno` nebo `obsazeno`.



Jak budeme řešit možné příchozí požadavky? Když někdo kabinku opustí, tak dostaneme její identifikátor, její index v poli. Můžeme na její místo tedy přímo zapsat, že je `volno`. To je jeden přímý krok, po kterém máme hotovo. Více formálně se bavíme o složitosti $\mathcal{O}(1)$.

Když někdo přijde a máme mu přidělit kabinku, už to tak rychlé nebude. Musíme pole projít a najít nějakou volnou. Můžeme začít například na prvním místě v poli a dokud nenarazíme na `volno`, hledáme dál. Pokud se dostaneme až na konec a žádné volno nebylo, ohlašujeme úplné zaplnění. A jak je to se složitostí? Očividně postup není tak přímočarý, kroků máme více. Když bude volno akorát v poslední kabince, budeme muset zkontrolovat všech N předchozích kabinek. Takže asymptotická složitost pro nejhorší případ je $\mathcal{O}(N)$. To vypadá, že máme stále prostor k zlepšování.

Co kdybychom si místo pole drželi seznam volných kabinek? Když někdo přijde, ze seznamu odstraníme libovolný prvek a vrátíme její identifikátor. Když někdo odejde, jeho kabinku na seznam připojíme. Na to se nám přesně hodí spojový seznam (můžeme na něj koukat i jako na zásobník). Všechny kroky pak uděláme v $\mathcal{O}(1)$, protože akorát přepojujeme pointery. Nikdy nebudeme muset procházet celý seznam. V případě, že je seznam prázdný, všechny kabinky jsou obsazené a při pokusu o získání kabinky hlásíme chybu.

⋆⋆⋆

Drobná praktická vložka – když píšeme programy, používáme k tomu paměť. A co je to paměť? Jen dlouhá řada kabinek, do kterých se dá schovat jeden byte. Moc víc za ní není.

Používáme-li například Cčko a chceme dynamickou paměť z haldy, musíme si o ni říct. Zavoláme funkci `malloc(size)`, které v argumentu předáme, kolik paměti (kolik kabinek) potřebujeme. Jako odpověď dostaneme identifikátor kusu paměti – pointer – ukazující na místo, které jsme dostali přiděleno. Nebo taky odpověď, že je plno a nic nedostaneme. Naopak, když už jsme hotoví, musíme paměť uvolnit. Zavoláme `free(void*)`. Funkci, které předáme dříve získaný pointer a tím paměť vrátíme na opakované použití. Nepřipomíná vám to něco?

Alokátor, který na tyto dotazy odpovídá, si na pozadí může udržovat datové struktury ne nepodobné tomu, co jsme teď navrhli. V případě zájmu se koukněte například na strukturu Free list.[3] Možných způsobů, jak naimplementovat alokátor je ale několik. Typicky má každá varianta nějaké výhody a nějaké nevýhody a nedá se říct, že by některé algoritmy byli všeobecně lepší než jiné. Už nás tam často nezajímá pouze asymptotická složitost, jde dost o rychlost na reálném hardwaru.

Úloha s alokováním paměti je obecně těžší než naše alokování kabinek, protože při ní alokujeme rozsahy, ne samostatné byty. Varianta odpovídající naší úloze se ale dá v praxi také najít při práci s bloky paměti o fixní velikosti. Například cache, kterou operační systém vytváří pro zrychlení přístupů k diskům. Nebo tzv. buffer pool u některých databází.

Vašek Šraier

Výsledková listina první série začátečnické kategorie 32. ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*Z1-1*	*Z1-2*	*Z1-3*	*Z1-4*	*Z1-5*	*Z1-6*	*série*	*celkem*
0.					8	10	10	12	12	14	66,0	66,0
1.	Eliška Macáková	CENADA BA	0	0	8	10	10	12	13	14	67,0	67,0
2.–6.	Robert Gemrot	GKomHavíř	3	0	8	10	10	12	12	14	66,0	66,0
	Jan Kotovský	GPísnickáPH	1	0	8	10	10	12	12	14	66,0	66,0
	Jakub Ondroušek	GTomkovaOL	0	0	8	10	10	12	12	14	66,0	66,0
	Darian Poljak	GJŠkodyPŘ	3	0	8	10	10	12	12	14	66,0	66,0
	Kateřina Vokálová	G Kolín	4	0	8	10	10	12	12	14	66,0	66,0
7.–9.	Michal Bravanský	GBílovec	2	0	8	10	10	12	12	12	64,0	64,0
	Vladimír Chudý	G Chrudim	3	0	8	8	10	12	12	14	64,0	64,0
	Kristýna Petrlíková	VOŠJičín	2	0	8	10	10	12	12	12	64,0	64,0
10.	Vít Ulehla	GJŠkodyPŘ	3	0	8	10	10	12	12	9	61,0	61,0
11.–12.	Martin Koňařík	GJŠkodyPŘ	3	0	8	10	10	12	12	7	59,0	59,0
	Ondřej Skácel	GTomkovaOL	1	0	8	10	10	12	12	7	59,0	59,0

[3] `https://en.wikipedia.org/wiki/Free_list`

	řešitel	*škola*	*ročník*	*sérií*	*Z1-1*	*Z1-2*	*Z1-3*	*Z1-4*	*Z1-5*	*Z1-6*	*série*	*celkem*
13.	Sebastián Svoboda	MendelGOP	3	0	8	10	10	12	10	7	57,0	57,0
14.	Patrik Herman	GTomkovaOL	1	0	8	10	10	1	12	14	55,0	55,0
15.	Petr Filip	GLovosice	1	0	8	10	10	12		14	54,0	54,0
16.	Tomáš Chabada	SPŠEMasLI	4	0	8	10	10	12	12		52,0	52,0
17.	Thomas Riedle	BRG APP	1	0	8	10	10		8	7	43,0	43,0
18.–27.	Jiří Bartošík	SUHr	3	0	8	10	10	12			40,0	40,0
	Jiří Bleha	SPSEPard	3	0	8	10	10	12			40,0	40,0
	Martina Daňková	GBystrc	3	0	8	10	10	12			40,0	40,0
	Janek Hlavatý	GJirsíkaČB	1	0	8	10	10	12			40,0	40,0
	Adam Húšťava	EupSchoolLux	2	0	8	10	10		12		40,0	40,0
	Ondřej Chlubna	GOrlová	3	0	8	10	10	12			40,0	40,0
	Robert Jaworski	GÚstavníPH	2	0	8	10	10	12			40,0	40,0
	Albert Kučera	GNadŠtolPH	3	0	8	10	10	12			40,0	40,0
	Matej Stencel	GPošKošice	3	0	8	10	10	12			40,0	40,0
	Vojtěch Žák	GŠpitálsPH	4	0	8	10	10	12			40,0	40,0
28.	Veronika Jůzková	MensaG	2	0	8	5	10		8		31,0	31,0
29.	Vojtěch Káně	G Brandýs	4	0	8	10		12			30,0	30,0
30.	Jakub Nevařil	G UherBrod	2	0	8	10	10	1			29,0	29,0
31.–40.	Martin Boček	MendelGOP	1	0	8	10	10				28,0	28,0
	Jan Hartman	GChodoviPH	4	0	8	10	10				28,0	28,0
	Klára Hloušková	G Kolín	4	0	8	1			12	7	28,0	28,0
	Vojtěch Máčala	G UherBrod	4	0	8	10	10				28,0	28,0
	Alexander Mateides	GJirsíkaČB	1	0	8	10	10				28,0	28,0
	Jan Najman	SPSEPard	3	0	8	10	10				28,0	28,0
	Jan Picek	GJírovcČB	3	0	8	10	10				28,0	28,0
	Ondřej Polanecký	SPŠPísek	3	0	8	10	10				28,0	28,0
	Štěpán Řihák	G UherBrod	4	0	8	10	10	0			28,0	28,0
	Daniel Šoltýs	GTřeKošice	2	0	8	10	10				28,0	28,0
41.	Jan Hlaváč	GNAlejíPH	4	0	8	10	6				24,0	24,0
42.	Martin Bencko	GOhradníPH	3	0	5	10	8				23,0	23,0
43.	Adam Krška	GMikulov	2	0	3	7		12			22,0	22,0
44.–45.	Kryštof Maxera	GJírovcČB	−1	0	6	10	4				20,0	20,0
	Patrik Baláš	SPSEPard	2	0		10	10				20,0	20,0
46.–48.	Šimon Andrš	GKepleraPH	1	0	8	10					18,0	18,0
	Martin Klimeš	GZábřeh	4	0	8	10					18,0	18,0
	Kristýna Umlaufová	SPŠOstrov	3	0	8	10	0				18,0	18,0
49.	David Maňásek	SUHr	3	0	5	2	10				17,0	17,0
50.–51.	Stanislav Müller	GČajOlom	1	0	8	5					13,0	13,0
	Erik Sabol	GČeskoliPH	0	0	8	5					13,0	13,0
52.–54.	Matěj Frič	GMatOS	4	0	8	2	0				10,0	10,0
	Filip Chytil	SPŠ Přerov	3	0	5	5					10,0	10,0
	Antonín Musil	PORGPha	3	0	8	2					10,0	10,0
55.	Matrix Svoboda	G UherBrod	4	0	8	1					9,0	9,0
56.–62.	Alexandr Čelakovský	G UherBrod	4	0	8						8,0	8,0
	Adam Ďubašík	G UherBrod	4	0	8						8,0	8,0
	Jiří Gallo	SPŠERožnov	3	0	8						8,0	8,0
	Petr Hýbl	G UherBrod	4	0	8						8,0	8,0
	Tadeáš Kozub	G UherBrod	4	0	8						8,0	8,0
	Arnošt Polák	PORG Krč	2	0	5					3	8,0	8,0
	Michal Zacek	MensaG	3	0	8						8,0	8,0
63.–66.	Marek Chwistek	MendelGOP	1	0	5						5,0	5,0
	Jan Machourek	GBBr	−1	0	5						5,0	5,0
	Kristýna Prokopová	GJosBožČT	4	0	5						5,0	5,0
	Tomáš Vesecký	SSŠVTPraha	3	0	5						5,0	5,0
67.	Jakub Sukdol	ZŠKubČB	−1	0		2,3					2,3	2,3
68.	Michal Koudela	G UherBrod	4	0	1						1,0	1,0