Za jak dlouho hlava přečte konkrétní úsek segmentů zjistíme pomocí výpočtu pro jednu hlavu. Tedy hlava buď nejdříve pojede k nejlevějšímu a pak k nejpravějšímu segmentu čteného úseku, nebo naopak. Celý algoritmus má časovou složitost $O(N)$, máme dohromady $N+1$ možností a výpočet každé z nich zabere konstantní čas. Pokud bychom souřadnice segmentů na vstupu nedostali setřízené, museli bychom je v čase $\mathcal{O}(N \log N)$ setřídit.

Paměťová složitost varianty pro jednu hlavu je konstantní (značíme $\mathcal{O}(1)$) – stačí nám si pamatovat pouze souřadnici nejlevějšího a nejpravějšího segmentu. Paměťová složitost varianty pro dvě hlavy je lineární (značíme $\mathcal{O}(N)$) – pamatujeme si N souřadnic segmentů a pak několik málo (konstantně) dalších pomocných proměných.

Pro lepší představu řešení nahlédněte do okomentovaného zdrojového kódu. Zdrojový kód je v jazyce C++, jeho znalost by však neměla být nutná pro pochopení programu.

Program C++:
`http://ksp.mff.cuni.cz/viz/26-Z2-6.cpp`

Karel Tesař

Výsledková listina druhé série začátečnické kategorie 26. ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*Z2-1*	*Z2-2*	*Z2-3*	*Z2-4*	*Z2-5*	*Z2-6*	*série*	*celkem*
0.					8	10	10	12	12	14	66,0	132,0
1.–2.	Václav Fabík	ZŠKřídloBO	−1	2	8	10	10	12	10	14	64,0	130,0
	Jan Tománek	GPelhřimov	3	2	8	10	10	12	12	14	66,0	130,0
3.	Jakub Pelc	G_UherBrod	0	2	8	10	10	12	12	14	66,0	129,0
4.	Miroslav Šerý	GValašKlob	1	2	8	10	10	12	10	13	63,0	123,5
5.	Jonáš Malena	SŠJeštědLI	4	2	8	10	10	12	10	12	62,0	119,5
6.	Přemysl Šťastný	GZamberk	0	2	8	10	10	12	12	11	63,0	116,0
7.	Václav Končický	GSOŠ_FrMís	3	2	8	10	10	12	12		52,0	104,0
8.	František Zajíc	G_Nymburk	1	2	8	10	10	12	4	6	50,0	102,0
9.	Lucie Studená	GKepleraPH	4	2	8	8	10	1	10	6	43,0	95,0
10.	Jakub Lukeš	GNAlejíPH	1	2	8	10	10	12	7	7	54,0	81,0
11.	Antonín Teichmann	GJeronýmLI	4	2	8	6			10		24,0	72,0
12.	Jiří Vozár	G_UherBrod	2	1							0,0	59,5
13.	Michal Převrátil	GKlatovy	1	2	8	10					18,0	58,0
14.	Michal Töpfer	G_DrJPekMB	1	1	8	10	10	12	7	6	53,0	53,0
15.	Marek Vitula	GJarošeBO	3	2	8	10					18,0	51,0
16.–17.	Jakub Heyduk	SŠP_ČB	4	2	8	10	10	12			40,0	48,0
	Petr Šíma	GKlatovy	1	2	8	10	10	12			40,0	48,0
18.	Pavel Mikuš	GMěl	3	2		10					10,0	46,0
19.	Lukáš Fruněk	GLesníZlín	1	1	8	10	10		7	5	40,0	40,0
20.–22.	Josef Gajdůšek	SŠKKamPard	1	1							0,0	39,0
	Václav Trpišovský	GOpenGaBab	−3	2					7	5	12,0	39,0
	Radovan Švarc	G_ČTřebová	3	2	1	10					11,0	39,0
23.	Milan Malina	GMikulášPL	1	2		6					6,0	30,0
24.	Jan Vargovský	GSPŠFrenšt	4	1							0,0	25,0
25.	Vojtěch Václavík	GSOŠ_FrMís	4	2	8						8,0	23,0
26.	Tomáš Michna	SPŠO	4	1	8	10	2				20,0	20,0
27.–29.	Štěpán Košan	GKlatovy	2	1	8	10					18,0	18,0
	Aneta Šťastná	GOmskPha	4	1	8	10					18,0	18,0
	Benedikt Žour	G_UherBrod	−1	1	8	6			4		18,0	18,0
30.–31.	Antonín Bruštík	G_UherBrod	3	2	8						8,0	17,0
	Jan Burda	G_Holice	−1	1	6	10	1	0			17,0	17,0
32.–35.	Jan Horák	GŠumperk	3	2	8	0					8,0	16,0
	David Karlík	G_UherBrod	3	2	8						8,0	16,0
	Viktor Kovařík	G_UherBrod	3	2	8						8,0	16,0
	Daniel Šerý	G_RožnovPR	2	1	8	8		0			16,0	16,0
36.	Zdeněk Pavlátka	GMikulášPL	2	1							0,0	15,5
37.	Martin Jílek	GKlatovy	2	1							0,0	13,0
38.–40.	Ivona Hrivová	GŽilina	4	2		6					6,0	11,0
	Dominik Krasula	GKrnov	1	1	1	10					11,0	11,0
	Jan Vozár	G_UherBrod	0	2	8						8,0	11,0
41.	Janek Hlavatý	ZŠ_DukelČB	−5	2	0	8					8,0	10,0
42.	Michaela Bačová	G_UherBrod	3	2	8						8,0	9,0
43.–44.	David Dvořáček	G_UherBrod	3	1							0,0	8,0
	Ivana Krumlová	GJarošeBO	1	1							0,0	8,0
45.	Petr Pacner	GBroumov	2	1							0,0	7,0
46.	Tereza Bohumská	GPisnickPH	−1	1							0,0	2,0
47.	Majtán Martin	G_SNPPiešť	1	1		0			1		1,0	1,0

Korespondenční Seminář z Programování

ZAČÁTEČNICKÁ KATEGORIE

26. ročník | KSP-Z | Březen 2014

Druhá série KSP-Z je úspěšně za námi a s ní pro vás máme jednu příjemnou novinku. Vzorová řešení KSP-Z se budeme snažit vydávat do týdne po uzávěrce série, abyste si je mohli přečíst, dokud ještě máte úlohy v živé paměti. Tady jsou a přejeme hodně štěstí s pochopením našich myšlenek!

Kdybyste některé řešení nemohli pochopit či potřebovali vysvětlit jakýkoli detail, nebojte se nás zeptat na našem fóru nebo emailu *ksp@mff.cuni.cz*.

Řešení druhé série začátečnické kategorie 26. ročníku KSP

26-Z2-1 Had z domina

Které kostky může Kevin otočit, aby součet čísel v obou řadách hada byl sudý? Nabízí se snadné dřevorubecké řešení: vyzkoušet všecky možnosti. To v tomto případě znamená zkoušet otáčet každou kostku a průběžně kontrolovat součty. Budou-li po otočení kostky součty teček v obou řadách sudé, přičteme k výsledku jedničku.

Takto jistě nalezneme správný výsledek, ale pro každou kostku musíme sečíst N čísel v horní i dolní řadě, a těch kostek je N. Asymptotická časová složitost je tedy $\mathcal{O}(N^2)$.

Had v zadání měl největší délku 1000, takže bylo potřeba nanejvýš řádově milion výpočtů. I tímto postupem se dalo dosáhnout plného počtu bodů. S tím se ale nespokojíme.

Představte si, že by Zuzka měla milion kostek místo tisíce, pak by takový výpočet na běžném počítači trval určitě více než čtvrt hodiny. A s většími vstupy ještě déle. Souvisí to s rychlostí různých algoritmů, dočtete se o nich v naší kuchařce o složitosti.[1]

Zkusme se na to podívat jinak. Zjistíme, že skácet celý les není potřeba, nemusíme počítat všechno. Úloha se dá řešit i rychleji.

Sudost nebo lichost čísla nazýváme stručně jedním slovem *parita*. Využijeme pozorování, že součet posloupnosti čísel má stejnou paritu jako součet jejich zbytků po dělení dvěma. Stačí nám tedy sečíst místo čísel ty zbytky a nakonec zjistit zbytek celého součtu po dělení dvěma. K tomu se hodí operace modulo. Výsledek bude ta parita. (Také můžeme modulit už během sčítání, při načtení sudého čísla paritu zachovat, při načtení lichého otočit, to je to samé.)

To také znamená, že když Kevin otočí kostku se dvěma sudými čísly, parity součtů sloupců se nezmění. Vymění se jenom nula za nulu. Stejně tak se dvěma lichými čísly. A naopak, když Kevin otočí kostku s jedním lichým a jedním sudým číslem, parity se otočí, protože v jedné řadě bude o jedničku míň a v druhé o jedničku víc. Sudé číslo plus nebo mínus jedna je vždy liché, z lichého se stane sudé.

Parity obou řad kostek si spočítáme jedním průchodem. Pokud mají obě řady na začátku sudou paritu (stručně řečeno parita řad je sudá-sudá), budeme otáčet kostky se stejnou paritou (říkejme jim „stejné“), aby se sudá-sudá zachovala. Jako výsledek tedy vypíšeme jejich počet. U lichá-lichá naopak budeme otáčet „různé“ kostky, tím nám vznikne sudá-sudá, a to přesně chceme.

Ještě může nastat možnost lichá-sudá a sudá-lichá. V takových případech nemáme co otáčet. Otáčením kostek umíme paritu buďto zachovat, nebo otočit. Z toho sudou-sudou nevyrobíme, proto vypíšeme jako výsledek nulu.

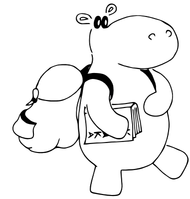
Kolik kostek je „stejných“ a kolik „různých“ si můžeme spočítat během jednoho průchodu vstupu zároveň s počítáním parit. Podle nich vypíšeme výsledek.

Vstup je na dvou řádcích. První řadu si tedy musíme uložit do paměti, abychom u čtení té druhé věděli, jaké bylo to první číslo na příslušné kostce. Paměťová složitost bude kvůli tomu lineární (stejně jako u dřevorubeckého řešení), časová ale jen $\mathcal{O}(N)$ místo $\mathcal{O}(N^2)$. Stačí nám totiž jen jeden průchod.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/26-Z2-1.py`

Program (C):
`http://ksp.mff.cuni.cz/viz/26-Z2-1.c`

Dominik Macháček



26-Z2-2 SADO

Nejprve si připomeneme některé užitečné matematické pojmy. Nejmenší společný násobek čísel a a b je nejmenší číslo takové, že ho lze beze zbytku vydělit a i b. Největší společný dělitel je naopak největší číslo takové, které beze zbytku dělí obě čísla. Jak na ně ale v programu přijít? Na výpočet největšího společného dělitele se používá *Euklidův algoritmus*. Kdo jej nezná, nechť se podívá do ukázkového kódu k této úloze. Jak funguje a jak je rychlý, se můžete dočíst v kuchařce o teorii čísel.[2]

Na výpočet nejmenšího společného násobku podobný algoritmus neexistuje, protože není potřeba. Platí totiž následující vztah:

$$a \cdot b = \mathrm{nsd}(a,b) \cdot \mathrm{nsn}(a,b)$$

[1] `http://ksp.mff.cuni.cz/viz/kucharky/slozitost`
[2] `http://ksp.mff.cuni.cz/viz/kucharky/teorie-cisel`

Na nejmenší společný násobek tedy přijdeme vydělením součinu čísel x a y na vstupu jejich největším společným dělitelem.

Kdo ovšem Euklidův algoritmus neznal, mohl na nejmenší společný násobek přijít jednoduše postupným zkoušením násobků čísla x, zda náhodou nejsou dělitelné y. To pro rychlé řešení úlohy bohatě stačilo.

A jak to celé souvisí se zadanou úlohou? Všechna čísla dělitelná zároveň x a y musí být nějakým násobkem nejmenšího společného násobku. A kolik takových čísel nalezneme v intervalu $[a, b]$ zjistíme třeba tak, že spočítáme počet násobků v intervalu $[0, b]$ a odečteme jejich počet v $[0, a-1]$. Obě hodnoty spočítáme jednoduchým dělením.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/26-Z2-2.py`

Ondra Hlavatý

26-Z2-3 Šifrovaná zpráva

Ke zdárnému vyřešení úlohy si budeme muset zkonstruovat nějakou *šifrovací tabulku*, která každému písmenu z rozsahu `A-Z` přiřadí odpovídající písmeno v zašifrovaném dopisu. Technicky to můžeme lehce zrealizovat pomocí jednoho pole o 26 prvcích – i-tý prvek bude vyjadřovat, na co se přeloží i-tý znak abecedy.

V ukázkovém programu využíváme toho, že písmena jsou v ASCII tabulce[3] umístěna za sebou a převod písmene na nějaké číslo uděláme jednoduše (bez nutnosti vědět, jaké přesně číslo v tabulce má) tím, že od něj odečteme hodnotu znaku `A`. Samotné `A` tak dostane hodnotu 0, `B` hodnotu 1 a tak dále. Jinou metodou je například použití *asociativního pole*, které nám poskytuje jazyk Python.

Když už máme technické detaily za sebou, stačí nám jen znak po znaku projít původní i zašifrovaný text a ke každému znaku původní zprávy uložit do šifrovací tabulky znak, na který je původní znak zašifrovaný. Toto by skvěle fungovalo, kdybychom měli slíbeno, že zpráva je vždy zašifrována správně. Jak však poznat chyby?

Důležité je uvědomit si, jaké chyby se nám mohou vyskytnout. Možné chyby jsou dvě. První z nich nastane, pokud stejné písmeno zašifrujeme dvakrát na písmena různá. To ale poznáme snadno, stačí nám přidat jednoduchou kontrolu u vyplňování naší šifrovací tabulky: Pokaždé, když budeme zpracovávat nějaký znak, se podíváme, jestli jsme mu už náhodou nepřiřadili nějaký znak dříve. Pokud ano a znaky jsou stejné, je vše v pořádku. Pokud se však znaky liší, zahlásíme chybu.

Druhá chyba nastane, pokud se dva různé znaky původní zprávy pokusíme zašifrovat na stejný znak zašifrované zprávy. V tomto místě nám pomůže další kontrola, ke které ale už budeme potřebovat druhou tabulku, říkejme jí třeba *tabulka použitých znaků*.

Vždy, když budeme přidávat nový záznam do šifrovací tabulky, poznačíme si do tabulky použitých znaků, že tento znak je již zabraný. Pokud pak narazíme na to, že námi chtěný znak již byl použit dříve, nezbude nám nic jiného, než také nahlásit chybu.

Mohlo by se zdát, že tím máme úlohu už zdárně vyřešenou a zašifrovanou zprávu zkontrolovanou. Ale ouha, zbývá nám ještě jedna drobnost na závěr – doplnit zbytek abecedy.

To je však už maličkost. Jen postupně projdeme šifrovací tabulku a u každého znaku, který doposud nemá přiřazený svůj zašifrovaný ekvivalent, najdeme první nepoužitý znak v tabulce použitých znaků. Ten přiřadíme a stejným způsobem doplníme celou abecedu.

Práce, kterou náš program musí udělat, je jednou projít původní i zašifrovaný text od začátku do konce a pak ještě celou abecedu. To, pokud si délku textu označíme jako N a velikost abecedy budeme brát jako malou konstantu, vede na celkovou časovou složitost $\mathcal{O}(N)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/26-Z2-3.c`

Jirka Setnička

26-Z2-4 Životně důležitá úloha

Připomeňme, že zadáním úlohy bylo v zadané posloupnosti o N číslech $A = a_0, a_1, \ldots, a_{N-1}$ nalézt všechny prvky s aritmetickým výskytem. Výskyt prvku x je aritmetický, pokud je posloupnost pozic, na kterých se prvek x v posloupnosti A nachází, aritmetická.

Prvky s aritmetickým výskytem mají být vypsány ve vzestupném pořadí a má u nich být uvedena i diference příslušné aritmetické posloupnosti. (Pokud má prvek jediný výskyt v celé posloupnosti A, chápeme jej jako prvek s aritmetickým výskytem o diferenci 0.)

Zdá se přirozené pevně si zvolit jeden prvek posloupnosti a ověřit, zda je jeho výskyt aritmetický. Pokud si však posloupnost A nepředzpracujeme, můžeme pro každý prvek projít celou posloupnost A (nebo její významnou část), což povede k řešení se složitostí $\mathcal{O}(N^2)$. S tou se nespokojíme.

Hodilo by se seskupit prvky posloupnosti A se stejnou hodnotou vedle sebe, abychom „aritmetičnost" výskytů ověřovali rychle. Pokud ale posloupnost A rovnou setřídíme, ztratíme informaci o pozicích prvků a nebudeme schopni aritmetičnost výskytu ověřit.

Tento problém snadno obejdeme – nebudeme třídit pouhé prvky posloupnosti A, ale budeme třídit posloupnost dvojic. K tomu účelu si zaveďme posloupnost $S = s_0, \ldots, s_{N-1}$, v níž $s_i = (a_i, i)$. Tedy i-tá dvojice udává hodnotu prvku na i-té pozici společně s touto pozicí.

Dvojice posloupnosti setřídíme podle slovníkového uspořádání (známé také pod cizojazyčným ekvivalentem *lexikografické*). Dvojice porovnáme podle první složky a v případě shody dále podle druhé. Například setříděním posloupnosti dvojic $(1,2),(0,1),(1,0),(1,1)$ lexikograficky bychom získali posloupnost $(0,1),(1,0),(1,1),(1,2)$.

Uvědomme si, že v okamžiku, kdy definujeme vlastní funkci pro porovnávání dvojic, neliší se třídění posloupnosti dvojic nijak od třídění posloupnosti čísel. Pokud s třídícími algoritmy nejste obeznámeni, nahlédněte do naší kuchařky.[4]

[3] `http://cs.wikipedia.org/wiki/ASCII`
[4] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

Když je posloupnost S lexikograficky setříděna, máme už výskyty každého prvku hned za sebou a ve vzestupném pořadí. Přímým průchodem pak ověříme, zda jsou posloupnosti výskytů jednotlivých prvků aritmetické.

Časová složitost třídění je $\mathcal{O}(N \log N)$, všechny ostatní části algoritmu už zvládneme v lineárním čase. Proto můžeme celkovou časovou složitost stanovit jako $\mathcal{O}(N \log N)$. Krom několika proměnných si stačí pamatovat pouze posloupnost S, proto je prostorová složitost lineární.

Závěrem si dovolíme ještě jeden mírně odlišný náhled řešení. Místo třídění posloupnosti dvojic můžeme od počátku udržovat pro každou hodnotu ze vstupní posloupnosti přihrádku. Při průchodu posloupnosti pak pro každý prvek rovnou vložíme jeho pozici do příslušné přihrádky. Na závěr zkontrolujeme, které přihrádky obsahují aritmetickou posloupnost.

Čísla v posloupnosti mohou dalece převyšovat délku posloupnosti, proto by přístup k jednotlivým přihrádkám přes pole nemusel být efektivní. Jak k nim přistupovat efektivně? K těmto účelům lze použít některou z datových struktur, které se skrývají pod souhrnným označením „asociativní pole".

Program C:
`http://ksp.mff.cuni.cz/viz/26-Z2-4.c`

Lukáš Folwarczný

26-Z2-5 Nedopité skleničky

Cílem úlohy bylo najít v setříděné posloupnosti dvě čísla (nemusí být nutně těsně za sebou), jejichž rozdíl se co nejvíce blíží číslu K. Nejjednodušším (ale pomalým) řešením tohoto problému by bylo projít každé dvě skleničky, odečíst jejich hladiny a porovnat velikost tohoto rozdílu s K.

Jak toto realizovat? Nejdřív se hodí si skleničky očíslovat. Ve většině programovacích jazyků má první prvek pole index 0, takže budeme skleničky stejně číslovat i my (od 0 do $N-1$). Dále si stačí pamatovat, jaký rozdíl mezi dvěma skleničkami byl nejblíže ke K (označme toto číslo MIN, na začátku nechť je v něm třeba ∞) a které dvě skleničky to byly (označme S_1, S_2). Pak pro každou skleničku vyzkoušíme všech $N-1$ dalších a při zkoušení každých takových dvou skleniček porovnáme, jestli je velikost rozdílu jejich obsahů ke K bližší než MIN. Pokud ano, aktualizujeme MIN i skleničky S_1 a S_2. Řekněme, že do S_1 uložíme číslo skleničky s menším obsahem šampaňského. Až projdeme všechny dvojice skleniček, bude v S_1 řešení, tedy číslo skleničky (počítáno od nulté), do které má Kevin nalít zbytek šampaňského.

Takové řešení prochází každou dvojici skleniček a provádí na nich porovnání. Proto má časovou složitost $\mathcal{O}(n^2)$. Jak jej zrychlit? Můžeme využít seřazenosti skleniček. Všimneme si, že pokud porovnáme i-tou a j-tou skleničku a rozdíl je už moc velký, tak rozdíl mezi i-tou a $(j+1)$–ní skleničkou bude ještě větší. Nemá tedy smysl porovnávat i-tou s $(j+1)$-ní a jakoukoliv další a můžeme místo toho i posunout o jedna dál a porovnávat $(i+1)$-ní a j-tou. Obdobně, jen obráceně, to funguje i v případě, že rozdíl je menší než K.

K realizaci použijeme dvě čísla (řekněme A a B), která budou označovat dvě aktuální pozice v posloupnosti skleniček. Tyto dvě skleničky označené čísly budeme v každém kroku porovnávat. Pokud budou skleničky očíslované od 0 do $N-1$, bude na počátku $A = 0$ a $B = 1$. Posloupnost skleniček si označíme jako s, tedy s_i bude množství tekutiny v i-té skleničce. Nyní můžeme procházet posloupnost s a hledat, pro která A a B bude $s_B - s_A$ co nejblíž ke K.

V každém kroku porovnáme $s_B - s_A$ s K. Pokud je $s_B - s_A > K$, zvětšíme A o 1, jinak zvětšíme B o 1. Zároveň pro každou dvojici kontrolujeme, jestli není lepší než doposud nejlepší řešení a upravujeme MIN, S_1 a S_2 – úplně stejně jako v pomalém řešení. Pokud $A = B$, tedy A „dohnalo" B, posuneme B o 1 doprava. Pokud $B > (N-1)$, prošli jsme všechny vhodné dvojice skleniček a můžeme skončit. V S_1 bude opět uloženo řešení.

Toto řešení v každém kroku posune A nebo B alespoň o 1, vždy platí $B \geq A$ a při $B > (N-1)$ skončí. Maximálně tedy udělá $2N$ kroků. Když zanedbáme onu dvojku, už víme, že má lineární časovou složitost $\mathcal{O}(N)$. Co se paměti týče, ukládáme konstantní počet čísel: A, B, MIN, S_1, S_2, N. Připočteno k velikosti vstupu načteného do paměti nám tedy vychází lineární paměťová složitost $\mathcal{O}(N)$.

Program (Python 2):
`http://ksp.mff.cuni.cz/viz/26-Z2-5.py`

Honza „Oggy" Škoda

26-Z2-6 Čtecí hlavy

Řešení této úlohy se řídí myšlenkou: „Pokud existuje jen málo možností, tak je vyzkoušíme všechny a z nich vybereme tu nejlepší." To je v informatice poměrně častý postup. Při aplikaci tohoto pravidla si však musíme dát pozor, obecně totiž může být zkoušení všech možností velmi neefektivní. Když se nám ale povede nalézt jen málo možností, mezi kterými určitě bude i ta úplně nejlepší, máme vyhráno.

Při čtení pomocí jedné hlavy máme pouze dvě možnosti. Buď hlava nejdříve pojede směrem k nejlevějšímu segmentu a pak směrem k nejpravějšímu segmentu, a nebo naopak nejdřív k nejpravějšímu a pak k nejlevějšímu. Z těchto možností vybereme tu lepší a máme výsledek. Časová složitost tohoto výpočtu je konstantní, ale časová složitost celého algoritmu je lineární (konkrétně $\mathcal{O}(N)$, kde N je počet čtených segmentů), protože musíme načíst vstup a zjistit, který segment je nejlevější a který nejpravější.

Při čtení pomocí dvou hlav máme také jen málo možností. Všimneme si, že v optimálním řešení levá hlava přečte nějaký počáteční úsek vybraných segmentů a pravá hlava přečte zbytek vybraných segmentů, které zas tvoří nějaký koncový úsek vybraných segmentů. Tyto dva úseky pokrývají dohromady všechny vybrané segmenty a nekříží se.

Bude mezi takovými řešeními určitě i to optimální? Ano, bude! Pokud by se obě hlavy měly po cestě křížit, tak namísto toho budeme uvažovat situaci, kdy se od sebe hlavy odrazí a to, co původně měla jet jedna hlava, pojede druhá hlava a naopak. Není tedy výhodné, aby si hlavy vyměnily pozice – levá hlava tedy celou dobu zůstane levou a pravá hlava pravou.

A co úsek čtený levou hlavou a úsek čtený pravou hlavou? Může být výhodné jejich překrytí? Také ne, protože nemá smysl, abychom nějaký segment četli dvakrát.

Stačí nám tedy vyzkoušet všechny možnosti. Pro souřadnice segmentů $s_1 < s_2 < \cdots < s_N$ vyzkoušíme možnosti, kdy levá hlava bude číst segmenty $s_1, \ldots, s_i$ a pravá hlava bude číst segmenty $s_{i+1}, \ldots, s_N$. Pozor, nesmíme zapomenout na možnosti, kdy levá resp. pravá hlava čte všechny segmenty.