69. ročník Matematické olympiády – 2019/2020

Úlohy krajského kola kategorie P

Krajské kolo 69. ročníku MO kategorie P se koná v úterý 21. 1. 2020 v dopoledních hodinách. Na řešení úloh máte 4 hodiny čistého času. V krajském kole MO-P se neřeší žádná praktická úloha, pro zajištění rovných podmínek řešitelů ve všech krajích je použití počítačů při soutěži zakázáno. Zakázány jsou rovněž jakékoliv další pomůcky kromě psacích potřeb (např. knihy, výpisy programů, kalkulačky, mobilní telefony). Řešení každé úlohy vypracujte na samostatný list papíru.

Řešení každé úlohy musí obsahovat:

- **Popis řešení,** to znamená slovní popis principu zvoleného algoritmu, *argumenty zdůvodňující jeho správnost* (případně důkaz správnosti algoritmu), diskusi o efektivitě vašeho řešení (časová a paměťová složitost). Slovní popis řešení musí být jasný a srozumitelný i bez nahlédnutí do samotného zápisu algoritmu (do programu). Není možné odkazovat se na vaše řešení úloh domácího kola, opravovatelé je nemusí mít k dispozici.
- **Zápis algoritmu.** Ve všech úlohách je třeba uvést zápis algoritmu v nějakém dostatečně srozumitelném pseudokódu (případně v programovacím jazyce Pascal, C/C++ nebo Python). Nemusíte detailně popisovat jednoduché operace jako vstupy, výstupy, implementaci jednoduchých matematických vztahů, vyhledávání v poli, třídění apod.

Za každou úlohu můžete získat maximálně 10 bodů. Hodnotí se nejen správnost řešení, ale také kvalita jeho popisu a efektivita zvoleného algoritmu. Algoritmy posuzujeme podle jejich časové složitosti, tzn. závislosti doby výpočtu na velikosti vstupních dat. Záleží přitom pouze na řádové rychlosti růstu této funkce. V zadání každé úlohy najdete přibližné limity na velikost vstupních dat. Efektivním vyřešením úlohy rozumíme to, že váš program spuštěný s takovými daty na současném běžném počítači dokončí výpočet během několika sekund.

Vzorová řešení úloh naleznete krátce po soutěži na webových stránkách olympiády *https://mo.mff.cuni.cz/*. Na stejném místě bude zveřejněn i seznam úspěšných řešitelů krajského kola a seznam řešitelů postupujících do ústředního kola.

Obraťte list.

P-II-1 Reverze

Na polici je vyznačeno n pozic a na každé z nich leží jedno jablíčko. Pozice jsou očíslovány zleva doprava čísly od 0 do $n-1$. Jablíčko na pozici i má velikost a_i. Všechna jablíčka mají navzájem různé velikosti.

Chceme všechna jablíčka uspořádat podle velikosti od nejmenšího po největší. Kvůli přísným hygienickým předpisům je však nesmíme přemisťovat sami, musíme na to použít techniku.

V místnosti s jablíčky máme mechanické rameno. Pomocí něho můžeme sebrat jablíčka z libovolného souvislého úseku police a vrátit je zpátky na polici na jejich původní místa, ale v opačném pořadí. Tuto operaci nazveme *reverze*. Každá reverze trvá stejně dlouho bez ohledu na to, jak dlouhý úsek obracíme.

Soutěžní úloha

Je zadáno počáteční rozmístění jablíček na polici. Napište program, který pomocí mechanického ramena jablíčka co nejefektivněji uspořádá.

Formát vstupu a výstupu

Na prvním řádku vstupu je jedno celé číslo n: počet jablíček.

Na druhém řádku vstupu jsou zadány velikosti jablíček: *navzájem různá* kladná celá čísla $a_0, \ldots, a_{n-1}$.

Na výstup vypište několik řádků – jeden pro každou reverzi, kterou chcete provést, a to v chronologickém pořadí. Pro každou reverzi vypište nejmenší a největší číslo pozice v úseku, který chcete obrátit.

Hodnocení

Hlavním kritériem hodnocení kvality vašeho řešení je *řádový* počet reverzí, které potřebujete provést. (Na konstantách nezáleží, takže například n^3 reverzí je stejně dobré řešení jako $7n^3 + 40$ reverzí.)

Sekundárním kritériem hodnocení je časová složitost vašeho programu.

Za řešení, kterému stačí provést lineární počet reverzí, ale má kvadratickou časovou složitost, můžete získat 6 bodů.

Za řešení, které potřebuje větší než lineární počet reverzí, můžete dostat nejvýše 4 body.

Příklady

Vstup:	*Výstup:*
7	4 6
10 50 40 30 70 60 20	1 4

Nejprve obrátíme úsek 4–6, čímž dostaneme pořadí jablíček 10 50 40 30 20 60 70. Potom obrátíme úsek 1–4 a tím dostaneme správné rostoucí pořadí.

Vstup:	*Výstup:*
7	5 6
20 30 40 50 60 70 10	4 5
	3 4
	2 3
	1 2
	0 1

Pro tento vstup existují i jiná řešení s menším počtem reverzí, než má řešení uvedené zde v příkladu.

P-II-2 Potrubí

Potřebujeme postavit potrubí délky *přesně* ℓ centimetrů, které bude mít *alespoň* f filtrů. Máme k dispozici několik kusů trubek. Každý kus má svou délku d_i a svůj maximální průtok za sekundu v_i. Některé trubky obsahují v sobě filtr, některé ne. Hodnoty ℓ a d_i jsou rozumně malá *kladná celá čísla.*

Soutěžní úloha

Zjistěte, zda je možné z trubek, které máme k dispozici, složit potrubí s požadovanými parametry. Pokud ano, určete také, jaký největší průtok může toto potrubí mít. (Průtok potrubí je roven minimu z průtoků trubek, z nichž se potrubí skládá.)

Formát vstupu a výstupu

Na prvním řádku vstupu jsou dána dvě celá čísla ℓ a f. Na druhém řádku vstupu je číslo n: počet trubek. Zbytek vstupu tvoří n řádků. Každý z nich je tvaru „d_i v_i f_i“ a popisuje jednu trubku (f_i je `A` pokud trubka obsahuje filtr, resp. `N`, pokud filtr neobsahuje).

Na výstup vypište maximální průtok hledaného potrubí, nebo -1, jestliže není možné požadované potrubí postavit.

Hodnocení

Plný počet bodů dostanete za řešení, které efektivně vyřeší vstupy s parametry $n, \ell, f \leq 10\,000$.

Nejvýše 7 bodů dostanete za řešení efektivní pro $n, \ell, f \leq 500$.

Nejvýše 5 bodů dostanete za řešení efektivní pro $n, \ell \leq 500$ a $f = 0$.

Nejvýše 3 body dostanete za řešení efektivní pro $n \leq 20$.

Příklady

Vstup:	*Výstup:*
100 0 4 40 1500 N 60 700 A 25 850 N 35 2700 A	850

Optimální je použít první, třetí a čtvrtou trubku. První a druhá trubka také tvoří potrubí správné délky, ale s menším průtokem.

Vstup:	*Výstup:*
200 2 3 100 1700 N 100 1500 A 100 1300 A	1300

Nemůžeme mít průtok vyšší než 1300, neboť musíme použít obě trubky s filtrem.

Vstup:	*Výstup:*
100 0 1 110 4747 A	-1

Jediná naše trubka je příliš dlouhá.

P-II-3 Virus

Tajný agent James v přestrojení za instalatéra úspěšně pronikl na nepřátelskou ambasádu. Na USB klíči si přinesl sofistikovaný virus, kterým by chtěl nakazit všechny počítače na ambasádě.

Na ambasádě je n počítačů. Některé dvojice počítačů jsou propojeny kabely a dokážou tak spolu přímo komunikovat. Takové dvojice nazveme sousední.

Běžné víry fungují tak, že nakažený počítač postupně nakazí všechny své sousedy. Jamesův virus je ale sofistikovanější. Aby se dalo obtížněji odhalit, odkud vlastně přišel, šíří se tak, že každý nakažený počítač postupně nakazí všechny *sousedy svých sousedů.*

Nakazit počítač vírem z USB klíče trvá dost dlouho a James při tom riskuje, že bude chycen a odhalen. Aby James minimalizoval riziko, chce nahrát virus do počítače jenom jednou.

Problém je, že při současném stavu propojení mezi počítači se může stát, že se virus nedokáže rozšířit na všechny ostatní počítače. James se proto rozhodl, že dříve než nějaký počítač nakazí, některé dvojice počítačů ještě propojí novými kabely. Také toto by ale bylo dobré provést co nejrychleji.

Soutěžní úloha

Je dán popis současného stavu počítačové sítě na ambasádě. Napište program, který připraví instrukce pro Jamese: nejprve najde jednu *nejmenší možnou* sadu nových kabelů, které má zapojit, a potom mu sdělí číslo počítače, na který má z USB klíče nahrát virus.

Formát vstupu a výstupu

Na prvním řádku vstupu jsou čísla $n \geq 3$ a m: počet počítačů na ambasádě a počet dvojic sousedních počítačů. Počítače mají čísla od 0 do $n-1$. Zbytek vstupu tvoří m řádků. Na každém z těchto řádků jsou vždy čísla dvou počítačů, které spolu sousedí (tj. mohou mezi sebou přímo komunikovat).

Na výstup postupně vypište popis všech kabelů, které má James přidat, a na závěr číslo počítače, na který má nahrát virus.

Hodnocení

Plný počet bodů dostanete za řešení, které efektivně vyřeší vstupy s $n \leq 100\,000$ a $m \leq 500\,000$.

Za řešení efektivní pro $n \leq 5000$ můžete dostat maximálně 7 bodů.

Nezapomeňte, že důležitou součástí řešení je důkaz jeho správnosti.

Příklady

Vstup:	*Výstup:*
4 6 0 1 0 2 0 3 1 2 1 3 2 3	0

Není třeba přidávat žádné kabely, stačí nakazit počítač číslo 0. Ten potom přímo nakazí všechny ostatní počítače. (Například počítač 1 je soused souseda počítače 0, protože 0 sousedí s 3 a 3 sousedí s 1.)

Vstup:	*Výstup:*
4 2 0 2 2 3	3 0 0 1 1

Až James natáhne dva nové kabely, bude počítač 1 sousedit s počítačem 0 a ten bude sousedit s počítači 2 a 3. Když tedy James nakazí počítač 1, tento počítač nakazí počítače 2 a 3. Následně jeden z těchto počítačů nakazí počítač 0. (Počítač 0 je sousedem souseda počítače 2, neboť 0 sousedí s 3 a 3 sousedí s 2.)

P-II-4 Exaktní exponenciální algoritmy

Tato soutěžní úloha navazuje na úlohu z domácího kola. Za zadáním úlohy najdete studijní text, který je totožný se studijním textem ze zadání domácího kola. Podúlohy jsou nezávislé, můžete je řešit v libovolném pořadí.

Podúloha A (5 bodů)

Máme n úkolů, které už měly být všechny hotové a je tedy potřeba vykonat je co nejdříve. Není možné dělat více úkolů najednou, je třeba provádět je postupně jeden po druhém. Práci na jednotlivých úkolech ovšem můžeme libovolně přerušovat a střídat (například nejprve 10 minut pracovat na prvním úkolu, potom 5 minut na druhém, potom 2 minuty nedělat nic, a pak opět 10 minut pracovat na prvním úkolu). Úkoly jsou očíslovány od 1 do n. Splnění úkolu číslo i trvá celkově t_i sekund.

Některé úkoly je vhodné dokončit ve správném pořadí. Přesněji řečeno, pro každou dvojici úkolů i, j je zadána nezáporná pokuta $s_{i,j}$, kterou musíme uhradit, když dokončíme úkol i dříve než úkol j.

Navíc platí, že úkoly je třeba splnit co nejdříve. Bude-li úkol i hotov po x sekundách od začátku, zaplatíme za něj pokutu $p(i, x)$. Pro každý úkol roste velikost pokuty v závislosti na čase, tzn. když $x_1 < x_2$, potom $p(i, x_1) < p(i, x_2)$. Pro různé úkoly mohou stejnému času dokončení odpovídat různě vysoké pokuty. Konkrétní podobu funkce $p(i, x)$ neznáte, můžete ji ale ve svém programu používat (jako kdybyste měli k dispozici knihovnu, která tuto funkci obsahuje).

Chceme vykonat všechny úkoly tak, aby byl součet pokut (dohromady za pořadí dokončení a za čas dokončení) co nejmenší. Navrhněte algoritmus s časovou složitostí $\hat{O}(2^n)$, který zjistí, jak toho dosáhnout. Nezapomeňte na důkaz správnosti.

Podúloha B (5 bodů)

Město je tvořeno n lokalitami a m ulicemi. Každá ulice je obousměrná a spojuje některé dvě lokality. V některých lokalitách bychom chtěli postavit veřejné záchody, a to tak, aby každá ulice měla alespoň na jednom konci záchod. Navrhněte co nejefektivnější algoritmus na určení nejmenšího počtu záchodů, které stačí postavit, když pro ně vhodně zvolíme lokality.

Při výpočtu časové složitosti nemusíte určovat přesné hodnoty konstant. Stačí například uvést složitost ve tvaru „$\hat{\mathcal{O}}(c^n)$, kde c je hodnota splňující následující vztah: . . . “.

Studijní text – Exaktní exponenciální algoritmy

Při analýze algoritmů se často setkáváme se zjednodušeným tvrzením, že algoritmy s *polynomiální* časovou složitostí považujeme za efektivní, zatímco algoritmy s *exponenciální* (a horší) časovou složitostí považujeme za neefektivní. V tomto ročníku olympiády trochu nahlédneme do světa exponenciálních algoritmů a uvidíme, že toto zjednodušení nemusí být vždy pravdivé.

Porovnání časových složitostí

Pro současné počítače můžeme odhadnout, že za minutu vykonají přibližně 10^{10} jednoduchých logických kroků programu. Máme-li tedy algoritmus s časovou složitostí $f(n)$ a zajímá nás, jak velké vstupy dokáže za minutu vyřešit, hledáme jednoduše největší n takové, že $f(n) \leq 10^{10}$. Výsledky pro některé zajímavé funkce uvádíme v tabulce:

$f(n)$	$n \log n$	n^2	n^3	$3n^4$	2^n	$1{,}42^n$	$1{,}1^n$	$n!$
$\max n$	500 000 000	100 000	2154	240	33	66	241	13

Vidíte, že například mezi polynomiální časovou složitostí $3n^4$ a exponenciální časovou složitostí $1{,}1^n$ není v praxi až tak velký rozdíl: oba algoritmy mají skoro stejné rozsahy efektivně řešitelných vstupů.

Možná jste si v tabulce všimli, že 66 je dvakrát 33. To není náhoda, platí totiž $(\sqrt{2})^n = \left(2^{1/2}\right)^n = 2^{n/2}$. Znamená to, že když časovou složitost algoritmu zlepšíme z 2^n na $\sqrt{2}^n \approx 1{,}42^n$, potom takto upravený algoritmus zvládne ve stejném čase vyřešit přibližně dvakrát větší vstup než původní algoritmus.

Tuto úvahu můžeme zobecnit. Výraz a^n upravíme následovně: platí $a = 2^{\log_2 a}$, a proto $a^n = \left(2^{\log_2 a}\right)^n = 2^{n \log_2 a}$. Tím například dostaneme, že $1{,}1^n$ je přibližně totéž jako $2^{0{,}1375n}$, resp. $2^{n/7{,}27254}$. Zlepšení časové složitosti z 2^n na $1{,}1^n$ tedy znamená, že novým algoritmem ve stejném čase vyřešíme více než sedmkrát větší vstup než algoritmem původním. Obecně platí, že každé snížení základu exponenciální funkce *několikrát* zvětší rozsah vstupů, které ještě dokážeme efektivně vyřešit.

Těžké problémy

V teoretické informatice máme mnoho algoritmických problémů, které jsou *těžké:* neznáme pro ně žádný algoritmus s polynomiální časovou složitostí a často máme dobré důvody domnívat se, že takové časové složitosti pro ně vůbec nelze dosáhnout. (Exaktní důkaz této domněnky pro jednu konkrétní sadu těžkých problémů je jedním z nejvýznamnějších otevřených problémů v informatice.)

Z pozorování uvedených v předchozí části studijního textu ovšem vyplývá jeden možný „směr útoku“: když narazíme na takovýto problém a potřebujeme ho exaktně vyřešit, jednou z možností je snažit se nalézt takový exponenciální algoritmus, jehož základ exponenciální funkce bude co nejmenší. Čím blíže k jedničce se dostaneme, tím větší vstupy dokážeme vyřešit v rozumném čase.

Ve studijním textu si ukážeme dva takové těžké problémy a předvedeme na nich dvě techniky návrhu šikovných exponenciálních algoritmů.

Zápis časové složitosti exponenciálních algoritmů

Při odvozování časové složitosti klasických efektivních algoritmů bývá zvykem zanedbávat konstanty. Místo přesného vyjádření, že algoritmus na vstupu velikosti n vykoná nejvýše $7n^2-3n+147$ kroků výpočtu, spokojíme se obvykle s asymptotickým odhadem „časová složitost algoritmu je $\mathcal{O}(n^2)$“ – neboli „časová složitost je nějaká funkce, která roste řádově nejvýše tak rychle, jako funkce n^2“.

Při analýze exponenciálních algoritmů někdy budeme podobným způsobem zanedbávat i polynomiální faktory. Takový horní odhad složitosti budeme zapisovat $\hat{\mathcal{O}}$. Například funkce $1{,}9^n$, $100 \cdot 2^n$ nebo $(3n^2+6)2^n + n^4$ patří obě do třídy $\hat{\mathcal{O}}(2^n)$, ale funkce $0{,}047 \cdot 2{,}01^n$ tam už nepatří.

Formálně, funkce f patří do $\hat{\mathcal{O}}(g)$ právě tehdy, když patří do $\mathcal{O}(p \cdot g)$ pro nějaký polynom p.

Časová složitost rekurzivních programů

Některé exaktní exponenciální algoritmy jsou založeny na *backtrackingu* (tzn. na rekurzivním prohledávání s návratem). Při analýze jejich časové složitosti budeme používat následující tvrzení:

Věta o časové složitosti rekurze. Mějme rekurzivní algoritmus A, který při řešení problému postupuje následovně: Když má vstup malé konstantní velikosti, vyřeší ho v konstantním čase. V obecném případě pro vstup velikosti n postupně provede k rekurzivních volání, přičemž při i-tém z nich se rekurzivně zavolá na vstup velikosti nejvýše $n - a_i$. (Hodnoty k a a_i jsou konstanty, které se během výpočtu nemění.) Kromě těchto rekurzivních volání algoritmus provede jenom polynomiálně mnoho kroků výpočtu vzhledem k n. Potom platí, že časová složitost algoritmu je $\hat{\mathcal{O}}(\alpha^n)$, kde α je jediné kladné reálné řešení rovnice

$$x^n - x^{n-a_1} - \cdots - x^{n-a_k} = 0.$$

Náčrt důkazu. Označíme-li časovou složitost našeho algoritmu T, z popisu algoritmu A dostáváme, že T splňuje rekurentní vztah $T(n) = T(n-a_1) + \cdots + T(n-a_k) + p(n)$, kde p je nějaký polynom. Když zanedbáme p a hledáme čistou exponenciální funkci T splňující tento rekurentní vztah, takže položíme $T(n) = \alpha^n$, dostaneme pro α výše uvedenou rovnici. Následně lze ukázat, že když za α vezmeme kladné reálné řešení této rovnice, potom náš algoritmus skutečně vykoná $\mathcal{O}(\alpha^n)$ rekurzivních volání. Celkový čas jeho běhu tedy můžeme shora odhadnout $\mathcal{O}(p(n) \cdot \alpha^n)$.

Příklady použití. Jestliže algoritmus při řešení problému velikosti n provede dvě rekurzivní volání na problémy velikosti $n-1$, dostáváme rovnici $x^n - x^{n-1} - x^{n-1} = 0$. Protože hledáme kladný reálný kořen, můžeme obě strany rovnice vydělit nenulovým výrazem x^{n-1} a dostaneme $x-1-1=0$, neboli $x=2$. Tento algoritmus má tedy časovou složitost $\hat{\mathcal{O}}(2^n)$.

Jestliže však algoritmus provede jedno rekurzivní volání na problém velikosti $n-1$ a jedno na problém velikosti $n-3$, dostaneme stejnou úvahou rovnici $x^3 - x^2 - 1 = 0$. Jejím jediným kladným reálným kořenem je $x \approx 1{,}4656$. Takový algoritmus má tedy časovou složitost $\hat{\mathcal{O}}(1{,}4656^n)$.

Maximální nezávislá množina

V zoologické zahradě právě postavili nový výběh. Mají n zvířat, která by do výběhu chtěli vypustit. Problém je ale v tom, že některé dvojice zvířat nemohou být spolu ve výběhu, neboť by se zvířata pokousala. Na vstupu dostanete seznam všech m takových dvojic. Navrhněte algoritmus, který zjistí, kolik nejvýše zvířat může skončit ve výběhu.

Dříve než se pustíme do lepších řešení, ukážeme si, jak lze tuto úlohu snadno vyřešit s časovou složitostí $\mathcal{O}(m2^n)$. Existuje přesně 2^n různých podmnožin zvířat. Postupně každou z nich vygenerujeme, projdeme celý seznam dvojic a podíváme se, zda jsme náhodou nevybrali obě zvířata z některé dvojice.

Maximální nezávislá množina: lepší algoritmus 1

Náš algoritmus bude mít podobu rekurzivní funkce, která dostane na vstupu nějakou množinu zvířat a na výstupu vrátí údaj, kolik nejvýše z těchto zvířat můžeme umístit do prázdného výběhu.

Uvažujme nějaké zvíře z. Optimální řešení, které *neobsahuje* z, najdeme tak, že se funkce rekurzivně zavolá na všechna zvířata kromě z. Jak najdeme optimální řešení, které *obsahuje* z? Označme $N(z)$ množinu těch zvířat, která nemohou být ve výběhu společně se zvířetem z. Když se rozhodneme do výběhu pustit zvíře z, zvířata z $N(z)$ tam pustit nemůžeme. Optimální řešení obsahující z tedy získáme tak, že se funkce rekurzivně zavolá na všechna zvířata kromě z a kromě množiny $N(z)$. Následně do takto získaného řešení přidáme ještě zvíře z. Na výstup naše funkce vrátí větší z obou právě popsaných řešení.

Je zjevné, že za zvíře z se vyplatí zvolit to zvíře, které má *co nejvíce* konfliktů s jinými – abychom při druhém rekurzivním volání dostali co nejmenší množinu zbývajících zvířat. Toto pozorování nás přivádí k následujícímu algoritmu:

1. Pokud má každé zvíře nejvýše jeden konflikt: Vezmi všechna bezkonfliktní zvířata. Z každé dvojice, která je v konfliktu, vezmi jedno libovolné zvíře. Tím výpočet končí.
2. V opačném případě najdi zvíře z, které má nejvíce konfliktů s ostatními zvířaty.
3. Rekurzivně najdi nejlepší řešení pro všechna zvířata kromě z.
4. Rekurzivně najdi nejlepší řešení pro všechna zvířata kromě z a $N(z)$, přidej do tohoto řešení z.
5. Na výstup vrať lepší z těchto dvou řešení.

Pro velké vstupy tento algoritmus vždy vykoná dvě rekurzivní volání. První je na vstup velikosti $n-1$. Jelikož vybrané zvíře z má alespoň dva konflikty, druhé rekurzivní volání je na problém velikosti nejvýše $n-3$. Z věty o časové složitosti rekurze tedy plyne, že toto řešení má časovou složitost $\hat{\mathcal{O}}(1{,}4656^n)$.

Maximální nezávislá množina: lepší algoritmus 2

Také tento algoritmus bude mít podobu rekurzivní funkce, která dostane na vstupu nějakou množinu zvířat a na výstupu vrátí údaj, kolik nejvýše z těchto zvířat

můžeme umístit do prázdného výběhu.

Připomeňme si, že optimální řešení, které *obsahuje* zvíře z, umíme nalézt tak, že najdeme optimální řešení pro všechna zvířata kromě z a $N(z)$ a potom do něj přidáme ještě zvíře z. Tentokrát budeme pokračovat trochu jinou myšlenkou. Tvrdíme, že v optimálním řešení, které z *neobsahuje,* musí být ve výběhu alespoň jedno ze zvířat patřících do $N(z)$. To je dost zjevné: řešení, v němž není ve výběhu ani z, ani žádné zvíře z $N(z)$, nemůže být optimální, neboť ho můžeme zlepšit přidáním zvířete z do výběhu.

Mějme následující algoritmus (slovo „nejméně“ v kroku 1 vysvětlíme později):

1. Najdi zvíře z, které má *nejméně* konfliktů s ostatními.
2. Pro každé zvíře y z množiny $\{z\} \cup N(z)$:
3. Rekurzivním voláním najdi nejlepší řešení pro všechna zvířata kromě y a $N(y)$.
4. Přidej do něj y, čímž dostaneš nejlepší řešení obsahující y.
5. Na výstup vrať nejlepší z řešení sestrojených v předcházejícím kroku.

Příklad. Nechť z je zebra a nechť zároveň s ní nemůže být ve výběhu kůň, srnka ani antilopa. Potom v optimálním řešení je alespoň jedno z těchto čtyř zvířat. Postupně tedy pro každé z nich najdeme rekurzivním voláním nejlepší řešení, které ho obsahuje.

Nechť má vybrané zvíře k konfliktů. Ze způsobu volby zvířete z v kroku 1 vyplývá, že *každé* zvíře má alespoň k konfliktů. Potom tento algoritmus provede $k+1$ rekurzivních volání, přičemž každé z nich bude na nějaký nový problém s nejvýše $n-(k+1)$ zvířaty.

Lze ukázat, že nejhorší případ nastane pro $k=2$, tedy když má každé zvíře přesně dva konflikty. V tomto případě bude mít tento algoritmus časovou složitost $\hat{\mathcal{O}}(3^{n/3})$, což můžeme upravit do podoby $\hat{\mathcal{O}}(1{,}4423^n)$.

Maximální nezávislá množina: nalezení všech optimálních řešení

„Lepší algoritmus 2“, který jsme právě popsali, můžeme snadno upravit tak, aby spočítal nejen největší počet zvířat ve výběhu, ale navíc aby postupně vygeneroval a vypsal všechna *optimální* řešení této úlohy. Z toho plyne, že optimálních řešení nemůže být více než $\hat{\mathcal{O}}(3^{n/3})$. Je jich tedy vždy výrazně méně než 2^n.

Snadno ukážeme, že tento odhad je poměrně těsný. Stačí vzít $n=3k$ zvířat, rozdělit je do trojic a říci, že v každé trojici jsou každá dvě zvířata v konfliktu. Potom bude každé optimální řešení obsahovat právě jedno zvíře z každé trojice, takže bude existovat přesně $3^k = 3^{n/3}$ optimálních řešení.

Problém obchodního cestujícího

V zemi se nachází n měst očíslovaných od 1 do n. Pro každou dvojici měst (i,j) známe cenu $c_{i,j}$ jízdenky z města i do města j. Obchodní cestující Emil potřebuje procestovat celou zemi: chce začít ve městě 1, postupně navštívit *právě jednou* každé jiné město a nakonec se vrátit zpět do města 1. Kolik peněz mu na to stačí?

Přímočaré řešení této úlohy má časovou složitost ještě horší než exponenciální. Emila zajímá, v jakém pořadí má navštívit města 2 až n, hledá tedy jejich optimální *permutaci*. To můžeme vyřešit tak, že postupně vygenerujeme všech $(n-1)!$ permutací měst 2 až n a pro každou z nich spočítáme, kolik by nás stálo jízdné. Takové řešení má časovou složitost $\hat{\mathcal{O}}(n!)$.

Problém obchodného cestujícího: dynamické programování

Ukážeme si, jak lze tuto úlohu vyřešit s časovou složitostí $\hat{\mathcal{O}}(2^n)$, přesněji v čase $\mathcal{O}(n^2 2^n)$.

Podívejme se na Emila někdy během jeho cesty. Už navštívil některá města a zaplatil nějaké peníze za jízdné. Položíme mu nyní otázku: „Za kolik nejméně peněz dokážeš svoji cestu dokončit?“

Na čem závisí odpověď? Pouze na dvou věcech: na městě a, kde se Emil právě nachází, a na množině měst B, která ještě nenavštívil. Označme $d_{a,B}$ odpověď na otázku s těmito dvěma parametry.

Je-li množina B prázdná, na otázku lze snadno odpovědět: $d_{a,\emptyset} = c_{a,1}$, neboť už se jenom potřebujeme vrátit z aktuálního města a na začátek. Ve všech ostatních případech se podíváme, co Emil udělá v následujícím kroku: vybere si některé město $b \in B$ a odcestuje do něj. Nejlepší řešení pro konkrétní město b bude Emila stát $c_{a,b} + d_{b,B\setminus\{b\}}$ peněz: nejprve zaplatí $c_{a,b}$ za cestu z a do b a potom $d_{b,B\setminus\{b\}}$ za optimální dokončení řešení v situaci, kdy stojí ve městě b a ještě potřebuje navštívit ostatní města z množiny B.

Hodnotu $d_{a,B}$ pro $B \neq \emptyset$ tedy spočítáme tak, že postupně vyzkoušíme všechna $b \in B$, pro každé z nich zjistíme, k jakému nejlepšímu řešení vede, a z takto získaných hodnot vezmeme minimum.

Zajímá nás celkem $\mathcal{O}(n2^n)$ různých hodnot $d_{a,B}$, neboť je n způsobů jak zvolit a a pro každé konkrétní a pak nejvýše 2^{n-1} možností pro B. Pro každé město kromě a máme totiž dvě možnosti: buď toto město v B leží, nebo tam neleží. Každou otázku dokážeme zodpovědět v čase $\mathcal{O}(n)$, takže celková časová složitost výpočtu všech hodnot $d_{a,B}$ je $\mathcal{O}(n^2 2^n)$. Výsledným řešením je potom hodnota $d_{1,\{2,3,\ldots,n\}}$.

Rekurzivní implementace vypadá takto:

Funkce $d(a, B)$:

1. Jestliže jsme už někdy zpracovali vstup (a, B):
2. Vrátíme zapamatovanou odpověď.
3. Jestliže je B prázdná:
4. *odpověď* $\leftarrow C[a, 1]$
5. Jinak:
6. *odpověď* $\leftarrow \min\{C[a, b] + d(b, B \setminus \{b\}) \mid b \in B\}$
7. Zapamatujeme si že pro vstup (a, B) je výstupem *odpověď*.
8. Vrátíme *odpověď*.

Všimněte si, že při každém rekurzivním volání se zmenší množina dosud nenavštívených měst. Díky tomu každá větev rekurze skončí. Pro každou z $\mathcal{O}(n2^n)$ dvojic

(a, B) se tělo této funkce (výpočet konkrétní hodnoty $d_{a,B}$) provede nejvýše jednou, proto skutečně dosáhneme slíbené celkové časové složitosti $\mathcal{O}(n^2 2^n)$.

Totéž můžeme zapsat bez rekurze:

```
1. Pro každé a:
2.     D[a, ∅] ← C[a, 1]
3.     Pro každou velikost vb množiny B od 1 do n − 1:
4.         Pro každou množinu B velikosti vb:
5.             Pro každé a ∉ B:
6.                 D[a, B] ← ∞
7.                 Pro každé b ∈ B:
8.                     D[a, B] ← min(D[a, B], C[a, b]+D[b, B\{b}])
```

Všimněte si, že v této implementaci při výpočtu nějaké hodnoty $d_{a,B}$ už známe všechny hodnoty $d_{b,B\setminus\{b\}}$, které potřebujeme, neboť jsme je vypočítali v dřívější iteraci vnějšího for-cyklu: množina $B \setminus \{b\}$ má menší velikost než množina B.

Na závěr dodejme, že při praktické implementaci tohoto algoritmu bychom pro uložení množin B použili tzv. bitové masky (bitmasky): množinu $\{x_1, \ldots, x_i\}$ bychom reprezentovali číslem $2^{x_1} + \cdots + 2^{x_i}$, tedy číslem, které má nastaveny právě bity s čísly $x_1, \ldots, x_i$.

Rozmyslete si, že má-li konkrétní množina přiřazeno nějaké číslo, potom všechny její podmnožiny mají menší čísla (neboť když smažeme z množiny prvek, tak ve dvojkovém zápisu čísla, které ji reprezentuje, změníme příslušnou jedničku na nulu). Místo vnějších dvou for-cyklů bychom tedy mohli použít jen jeden for-cyklus přes všechna čísla představující platné kódy množin, od nejmenšího po největší. Tím dostaneme jiné pořadí, v němž budeme množiny B zpracovávat, ale výše popsaný algoritmus bude stále korektně fungovat, protože pro každé B a b bude i nyní platit, že množinu $B \setminus \{b\}$ zpracujeme dříve než množinu B.