69. ročník Matematické olympiády – 2019/2020

Úlohy domácího kola kategorie P

Úlohy P-I-1 a P-I-2 jsou praktické, vaším úkolem v nich je vytvořit a odladit efektivní program v jazyce Pascal, C nebo C++. Řešení těchto dvou úloh odevzdávejte ve formě zdrojového kódu přes webové rozhraní přístupné na stránce *https://mo.mff.cuni.cz/submit/,* kde také naleznete další informace. Odevzdaná řešení budou automaticky vyhodnocena pomocí připravených vstupních dat a výsledky vyhodnocení se dozvíte krátce po odevzdání. Pokud váš program nezíská plný počet 10 bodů, můžete své řešení opravit a znovu odevzdat.

Úlohy P-I-3 a P-I-4 jsou teoretické, za každou z nich lze získat až 10 bodů. Řešení musí obsahovat popis algoritmu, zdůvodnění jeho správnosti a odhad časové a paměťové složitosti. Nemusíte psát program, algoritmus stačí zapsat ve vhodném pseudokódu nebo dokonce jenom slovně, ale v tom případě dostatečně podrobně a srozumitelně. Hodnotí se nejen správnost, ale také efektivita zvoleného postupu řešení. Úloha P-I-4 je tvořena třemi samostatnými podúlohami. Část bodů dostanete, i když vyřešíte správně jenom některou z nich. Řešení obou teoretických úloh odevzdávejte ve formě souboru typu PDF přes výše uvedené webové rozhraní.

Řešení všech úloh můžete odevzdávat do 15. listopadu 2019. Opravená řešení a seznam postupujících do krajského kola najdete na webových stránkách olympiády na adrese *https://mo.mff.cuni.cz/,* kde jsou také k dispozici další informace o kategorii P.

Experimentální jazyky: V letošním ročníku MO-P jsme experimentálně povolili odevzdávat praktické úlohy i v jazycích Python 3 a Java 11. U nich nicméně nezaručujeme, že bude možné získat plný počet bodů – je možné, že program nestihne doběhnout do časového limitu, byť používá algoritmus s optimální časovou složitostí. Také neslibujeme, že tyto jazyky budou k dispozici v dalších kolech soutěže.

P-I-1 Schodiště

Na horu Inari vede dlouhatánské nepravidelné schodiště. Jeho schody jsou očíslovány zdola nahoru od 1 do n. Výšku schodu i označíme h_i.

Mnich Takeši každé ráno vynáší vědro vody po schodišti na horu Inari. Začíná u studně vykopané těsně pod schodem 1 a končí až ve svatyni zbudované na vrcholu hory hned nad schodem n.

Takeši každým krokem stoupá alespoň o jeden schod výše, občas ale udělá delší krok a některé schody vynechá. Největší výškový rozdíl, který zvládne překonat jedním krokem, označíme d. Takový krok dokáže udělat bez ohledu na to, kolik schodů při tom vynechá.

Pro zadaný popis schodiště vypočítejte, kolika různými způsoby může Takeši vystoupat na horu.

Formát vstupu a výstupu

Na prvním řádku vstupu je uveden počet schodů n a maximální výška kroku d.

Na druhém řádku vstupu jsou celá čísla $h_1, \ldots, h_n$ udávající výšky jednotlivých schodů.

Na výstup vypište jeden řádek s jedním celým číslem: zbytek, který dává hledaný počet způsobů po dělení číslem $10^9 + 7$.

Omezení a hodnocení

Vaše řešení bude testováno na pěti sadách vstupních dat, za správné vyřešení každé z nich dostanete dva body. V každé sadě platí $1 \leq h_i \leq 10^9$ a také $h_i \leq d \leq 10^9$ pro všechna i. V jednotlivých sadách platí následující další omezení:

- Sada #1: $n \leq 16$.
- Sada #2: $n \leq 50$ a v každém vstupu existuje nejvýše 10^6 různých způsobů výstupu na horu.
- Sada #3: $n \leq 1000$.
- Sada #4: $n \leq 100\,000$ a $d \leq 10$.
- Sada #5: $n \leq 500\,000$.

Příklad

Vstup:	*Výstup:*
`4 100` `20 30 50 30`	`6`

Číslem i označíme stav, v němž Takeši právě stojí na i-tém schodu – speciálně 0 označuje stav, kdy ještě nezačal stoupat po schodech. Existuje šest různých způsobů, jak může Takeši vystoupat na horu. Odpovídají jim následující posloupnosti stavů: 01234, 0124, 0134, 0234, 024 a 034.

P-I-2 Nová bankovka

V Kocourkově mají systém platidel velmi podobný našemu. Nejmenší nominální hodnotou je 1 tolar. Existují mince a bankovky s následujícími pěknými, systematicky zvolenými nominálními hodnotami: 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10 000, 20 000 a 50 000 tolarů.

Kocourkovská centrální banka se rozhodla, že vydá ještě jednu novou bankovku v hodnotě přesně x tolarů.

Obyvatelé Kocourkova si nyní kladou řadu otázek tohoto typu: „Když budu chtít zaplatit přesně t_i tolarů, kolik nejméně kusů platidel stačí použít?“

Formát vstupu a výstupu

Na prvním řádku vstupu je číslo x: nominální hodnota nové bankovky. Na druhém řádku je číslo q: počet otázek, které mají obyvatelé Kocourkova. Na třetím řádku jsou mezerami oddělená čísla $t_1, \ldots, t_q$: sumy pro jednotlivé otázky.

Na výstup vypište q řádků: postupně pro každou otázku jeden řádek s příslušnou správnou odpovědí.

Omezení a hodnocení

Vaše řešení bude testováno na pěti sadách vstupních dat, za správné vyřešení každé z nich dostanete dva body. V každé sadě platí $q \leq 50$ a je zaručeno, že x bude různé od nominálních hodnot existujících platidel. V jednotlivých sadách platí následující další omezení:

sada	x	*všechna* t_i
#1	≤ 20	≤ 20
#2	$= 100\,000$	$\leq 10^5$
#3	$\leq 10^5$	$\leq 10^5$
#4	$\leq 10^7$	$\leq 10^7$
#5	$\leq 2 \cdot 10^9$	$\leq 2 \cdot 10^9$

Příklad

Vstup:	*Výstup:*
4700	3
4	2
53 9400 9401 30000	3
	2

Nová bankovka má hodnotu 4700 tolarů.

- *Nejlepším způsobem jak zaplatit 53 tolarů je použít platidla s hodnotami 50 + 2 + 1.*
- *Nejlepším způsobem jak zaplatit 9400 tolarů je použít dvě nové bankovky.*
- *Nejlepším způsobem jak zaplatit 9401 tolarů je použít dvě nové bankovky a jednu 1-tolarovou minci.*
- *Nejlepším způsobem jak zaplatit 30000 tolarů je 10000 + 20000.*

P-I-3 Rekonstrukce mapy

Cestovatel Parko Molo nedávno navštívil jedno ostrovní království tvořené n stejně vypadajícími ostrovy. Některé dvojice ostrovů byly propojeny mosty, a to tak, že po mostech bylo možné dojet z libovolného ostrova na libovolný jiný ostrov právě jedním způsobem. (Mostů tedy bylo přesně $n - 1$ a odborně říkáme, že království mělo stromovou topologii.)

Když už Parko z království odcestoval, uvědomil si, že úplně zapomněl nakreslit si jeho mapu. Ve svém notesu našel jenom seznam, kam si zapsal, kolik mostů vedlo ze kterého ostrova. Ale ani správnosti tohoto seznamu úplně nedůvěřuje.

Soutěžní úloha

Jsou dána čísla $d_1, \ldots, d_n$. Zjistěte, zda existuje ostrovní království výše popsaných vlastností, jehož ostrovy je možné očíslovat od 1 do n tak, aby pro každé i platilo, že z ostrova i vede přesně d_i mostů.

Pokud takové království existuje, sestrojte jeho (jednu možnou) mapu – přesněji seznam mostů spojujících jeho ostrovy.

Formát vstupu a výstupu

Na prvním řádku vstupu je uvedeno kladné celé číslo n. Na druhém řádku vstupu jsou mezerami oddělená kladná celá čísla $d_1, \ldots, d_n$. (Tato čísla se vejdou do běžné celočíselné proměnné, ale nic jiného o jejich velikosti nepředpokládejte.)

Pokud takové království neexistuje, vypište na výstup jeden řádek s řetězcem „`neexistuje`“.

Pokud takové království existuje, výstup bude tvořen $n - 1$ řádky a na každém z nich budou dvě celá čísla z rozsahu od 1 do n: čísla dvou ostrovů spojených mostem. Tato čísla musí být zvolena tak, aby se z každého ostrova dalo dojet po mostech na libovolný jiný ostrov a navíc aby platilo, že z každého ostrova i vede přesně d_i mostů. Existuje-li více různých řešení, vypište jedno libovolné z nich.

Omezení a hodnocení

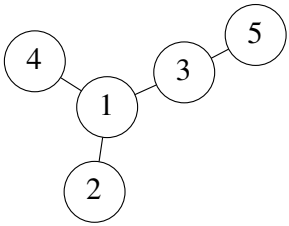
- Za řešení s optimální časovou složitostí můžete dostat plný počet 10 bodů.
- Za jiné řešení, které je dostatečně efektivní na to, aby na běžném počítači během několika sekund vyřešilo vstup velikosti $n = 10^6$, můžete dostat až 8 bodů.
- Řešení dostatečně efektivní pro $n = 5\,000$ bude ohodnoceno až 6 body.
- Za libovolné funkční řešení lze získat 3 body, bez ohledu na jeho efektivitu.
- Nezapomeňte, že důležitou součástí řešení je důkaz jeho správnosti.

Příklady

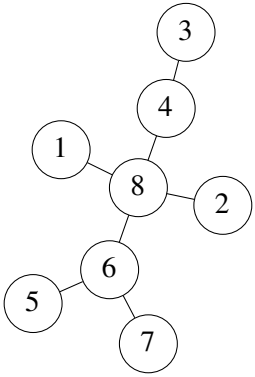
Vstup:	*Výstup:*
3 4 1 2	neexistuje

Rozmyslete si, že v žádném království nemohou být dva ostrovy přímo spojeny více než jedním mostem. Máme-li tedy pouze tři ostrovy, nemůže existovat ostrov, z něhož vedou čtyři mosty.

Vstup:	*Výstup:*
5 3 1 2 1 1	1 3 1 2 1 4 3 5



Vstup:	*Výstup:*
8 1 1 1 2 1 3 1 4	1 8 2 8 3 4 4 8 5 6 6 7 6 8



P-I-4 Exaktní exponenciální algoritmy

K této úloze se vztahuje studijní text uvedený na následujících stranách. Doporučujeme vám nejprve prostudovat studijní text a až potom se vrátit k samotným soutěžním úlohám.

a) *(2 body)* V úloze o maximální nezávislé množině uvažujme nyní jenom vstupy, v nichž platí, že každé zvíře má nejvýše dva konflikty. Navrhněte algoritmus s polynomiální časovou složitostí, který pro libovolný takový vstup vypočítá, kolik nejvýše zvířat můžeme pustit do výběhu.

b) *(4 body)* Ukažte, jak lze pomocí algoritmu z podúlohy a) zlepšit „lepší algoritmus 1“ ze studijního textu. Jakou časovou složitost bude mít algoritmus, který takto dostanete?

c) *(4 body)* Máme n krabic. Každá krabice má nějakou výšku (v cm), nějakou hmotnost a nějakou nosnost (oboje v kg). Z některých krabic chceme postavit věž: jednu krabici položíme na zem, druhou na první, třetí na druhou, a tak dále. Krabice můžeme na věž pokládat v libovolném pořadí. Věž je stabilní, pokud pro každou krabici platí, že součet hmotností všech krabic umístěných nad ní je menší nebo roven její nosnosti.

Navrhněte algoritmus s časovou složitostí $\hat{\mathcal{O}}(2^n)$, který zjistí, jakou nejvyšší stabilní věž můžeme z dané sady krabic postavit.

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. $D[a, \emptyset] \leftarrow C[a, 1]$
3. Pro každou velikost vb množiny B od 1 do $n - 1$:
4. Pro každou množinu B velikosti vb:
5. Pro každé $a \notin B$:
6. $D[a, B] \leftarrow \infty$
7. Pro každé $b \in B$:
8. $D[a, B] \leftarrow \min(D[a, B], C[a, b]+D[b, B\setminus\{b\}])$

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.