69. ročník Matematické olympiády – 2019/2020

Úlohy ústředního kola kategorie P – 1. soutěžní den

Na řešení úloh máte 5 hodin čistého času. Řešení každé úlohy odevzdejte do soutěžního systému jako samostatný soubor.

Řešení každé úlohy musí obsahovat:

- **Popis řešení,** to znamená slovní popis principu zvoleného algoritmu, argumenty zdůvodňující jeho správnost, 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 povoleno odkazovat se na vaše řešení předchozích kol, opravovatelé je nemají k dispozici; na autorská řešení se odkazovat můžete.
- **Zápis algoritmu.** Ve všech úlohách je třeba uvést zápis algoritmu, a to buď ve tvaru zdrojového textu nejdůležitějších částí programu v jazyce Pascal nebo C/C++, nebo v nějakém dostatečně srozumitelném pseudokódu. 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 (včetně zdůvodnění správnosti) a efektivita zvoleného algoritmu. Algoritmy posuzujeme zejména podle jejich časové složitosti, tzn. podle 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.

P-III-1 Oplocení zahrádek

Dědici pozemkového magnáta Pišišvora se rozhodli rozdělit získaný obdélníkový pozemek pomocí plotů na čtvercové zahrádky se stranou délky jeden metr a tyto zahrádky rozprodat. Problém spočívá v tom, že v této oblasti je stavění plotů přísně regulováno, a to následovně:

- Plot musí být rovný.
- Plot musí být rovnoběžný s jednou ze stran pozemku a musí mít od ní celočíselnou vzdálenost v metrech.
- Ploty se musí stavět postupně jeden po druhém.
- Plot nesmí křižovat jiné, dříve postavené ploty.
- Za každý plot je třeba zaplatit registrační poplatek. Tento poplatek je jednoznačně určen souřadnicí, na které celý plot leží. Poplatek nezávisí na délce plotu.
- Každý plot musí být co nejdelší, musí tedy začínat i končit v bodě, kde narazí na jiný (na něj kolmý) plot.

Celý obvod pozemku je už oplocen, uvnitř zatím žádné ploty nejsou.

Formát vstupu a výstupu

Na prvním řádku vstupu jsou celá čísla d, s udávající délku a šířku obdélníkového pozemku.

Na druhém řádku je $d-1$ celých čísel $z_1, \ldots, z_{d-1}$, kde z_i je poplatek za postavení každého „svislého" plotu, tedy plotu, jehož každý bod je vzdálen i metrů od levého okraje pozemku.

Na třetím řádku je $s-1$ celých čísel $v_1, \ldots, v_{s-1}$, udávajících ceny za „vodorovné" ploty podle jejich vzdálenosti od horního okraje pozemku.

Vypočítejte a na výstup vypište minimální celkovou cenu, za kterou můžeme postavit ploty tak, aby každé políčko velikosti metr krát metr bylo ze všech stran oploceno.

Omezení a hodnocení

Všechny poplatky jsou kladné. Cena libovolného korektního oplocení se bez problémů vejde do běžné celočíselné proměnné.

Za libovolné správné řešení můžete získat 3 body.

Za řešení dostatečně efektivní pro $d, s \leq 50$ dostanete 6 bodů.

Omezení pro optimální řešení úmyslně neuvádíme. Upozorňujeme však, že u řešení, která mají ambici získat více než 6 bodů, je velmi důležitou součástí důkaz správnosti použitého algoritmu.

Příklad

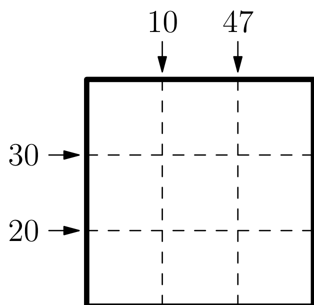
Vstup:

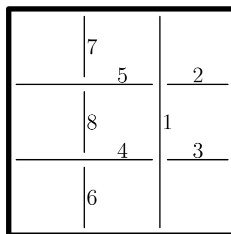
```
3 3
10 47
30 20
```

Výstup:

```
177
```

Na obrázku vlevo je znázorněno zadání. Čárkované čáry představují místa, kde je třeba postavit ploty. Čísla udávají ceny za postavení každého plotu v dotyčném řádku/sloupci. Na obrázku vpravo je znázorněno jedno možné optimální řešení. Čísla udávají pořadí stavění plotů.





P-III-2 Horolezci

Horolezecký klub chystá n-denní expedici. V každém dnu trvání expedice se jeden z jejích účastníků pokusí zdolat jeden z okolních vrcholů. Vrchol plánovaný na i-tý den je vysoký v_i metrů.

Každý horolezec, který bude vystupovat alespoň na jeden vrchol, potřebuje mít speciální výstroj. Cena takové výstroje je přímo úměrná maximální výšce, pro kterou je výstroj certifikována. Pro jednoduchost budeme předpokládat, že výstroj certifikovaná pro výstupy až do výšky x stojí x korun.

Horolezec může během expedice použít svoji výstroj opakovaně vícekrát. Kvůli bezpečnosti členů expedice ale platí následující omezení:

- Horolezec nesmí vystoupit do výšky, na kterou nemá certifikovanou výstroj.
- Každý horolezec musí používat pouze svoji vlastní výstroj, nesmí si výstroj půjčit od druhého.
- Po každém pokusu o dosažení vrcholu musí horolezec alespoň den odpočívat.

Jsou dány výšky v_i. Navrhněte algoritmus, který spočítá minimální celkovou cenu potřebného vybavení pro expedici. Hledáme tedy, kolik členů má mít expedice, jakou výstroj máme kterému z nich koupit a kdo z nich bude stoupat na vrchol ve který den. Toto celé chceme zorganizovat tak, aby celková cena potřebných horolezeckých výstrojí byla co nejmenší.

Formát vstupu a výstupu

Na prvním řádku vstupu je kladné celé číslo n. Na druhém řádku vstupu jsou kladná celá čísla $v_1, \ldots, v_n$.

Na výstup vypište minimální celkovou cenu horolezeckých výstrojí, s nimiž lze zvládnout expedici.

Omezení a hodnocení

Součet všech výšek se vejde do běžné celočíselné proměnné.

Za libovolné správné řešení můžete získat 3 body. Za řešení efektivní pro $n \leq 100$ dostanete 6 bodů, za řešení efektivní pro $n \leq 1000$ dostanete 8 bodů, za řešení efektivní pro $n \leq 30\,000$ plných 10 bodů.

Příklady

Vstup:	*Výstup:*
`5` `8485 8516 8586 8611 8848`	`17459`

Nejvyšších pět osmitisícovek. Optimálním řešením je mít horolezce A s výstrojí certifikovanou do 8848 metrů a horolezce B s výstrojí do 8611 metrů. Pokusy o dosažení vrcholu budou provádět v pořadí ABABA.

Vstup:	*Výstup:*
`7` `1200 7500 1100 1500 1400 1100 7300`	`10000`

Jedno možné optimální řešení: Horolezci A (výstroj do 1100), B (výstroj do 7500) a C (výstroj do 1400) půjdou lézt v pořadí CBABCAB.

P-III-3 Exaktní exponenciální algoritmy

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

Podúloha A (5 bodů)

Připomeňme si, že v zoologické zahradě ze studijního textu mají n zvířat, přičemž existuje m dvojic zvířat, která nemohou být společně ve výběhu. Pro daná n, m a seznam dvojic zvířat chceme nyní zjistit, zda je možné všechna tato zvířata rozmístit do tří výběhů. Navrhněte co nejefektivnější algoritmus řešící tuto úlohu.

Za řešení s časovou složitostí $\hat{\Theta}(b^n)$ můžete získat nejvýše 5 bodů pro $b < 2$, nejvýše 3 body pro $b = 2$ a nejvýše 1 bod pro $b > 2$.

Podúloha B (5 bodů)

Vstup je stejný jako v podúloze A. Navrhněte algoritmus, který určí, kolik nejméně výběhů potřebujeme, chceme-li každé zvíře umístit do některého výběhu.

Plných 5 bodů lze získat za řešení s časovou složitostí $\hat{\Theta}(b^n)$, kde $b < 3$. Za libovolné řešení s exponenciální časovou složitostí můžete dostat alespoň 3 body. Za řešení s časovou složitostí horší než exponenciální vzhledem k n dostanete nejvýše 1 bod.

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

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.