Priebeh krajského kola

Krajské kolo 35. ročníka Olympiády v informatike, kategória A, sa koná 21. januára 2020 v dopoludňajších hodinách. Na riešenie úloh majú súťažiaci **4 hodiny čistého času**. Rôzne úlohy riešia súťažiaci na samostatné listy papiera. Akékoľvek pomôcky okrem písacích potrieb (napr. knihy, výpisy programov, kalkulačky) sú zakázané.

Čo má obsahovať riešenie úlohy?

- Slovne popíšte algoritmus.
  Slovný popis riešenia musí byť jasný a zrozumiteľný i bez nahliadnutia do samotného algoritmu/programu.
- Zdôvodnite správnosť vášho algoritmu.
- Uveďte a zdôvodnite jeho časovú a pamäťovú zložitosť.
- Podrobne uveďte dôležité časti algoritmu, ideálne vo forme programu v nejakom bežnom programovacom jazyku (napr. C++, Python, Java, Pascal).
- V prípade, že používate vo svojom programovacom jazyku knižnice, ktoré obsahujú implementované dátové štruktúry a algoritmy (napr. STL pre C++), v popise algoritmu stručne vysvetlite, ako by ste napísali program s rovnakou časovou zložitosťou bez použitia knižnice.

Hodnotenie riešení

Za každú úlohu môžete získať od 0 do 10 bodov.

Pokiaľ nie je v zadaní povedané ináč, najdôležitejšie dve kritériá hodnotenia sú v prvom rade **správnosť** a v druhom rade **efektívnosť** navrhnutého algoritmu. Na výslednom počte bodov sa môže prejaviť aj kvalita popisu riešenia a zdôvodnenie tvrdení o jeho správnosti a efektívnosti.

Efektívnosť algoritmu posudzujeme vypočítaním jeho časovej zložitosti – funkcie, ktorá hovorí, ako dlho vykonanie algoritmu trvá v závislosti od veľkosti vstupných parametrov. Nezávisí pri tom na konštantných faktoroch, len na rádovej rýchlosti rastu tejto funkcie.

V zadaní úloh uvádzame časť „Hodnotenie“, v ktorej nájdete približné limity na veľkosť vstupných údajov. Pod pojmom „efektívne vyriešiť“ chápeme to, že váš program spustený na modernom počítači by mal dať odpoveď nanajvýš do niekoľkých sekúnd.

Údaje z tejto časti zadania by mali slúžiť hlavne na to, aby ste o riešení, ktoré vymyslíte, vedeli približne povedať, koľko bodov zaň dostanete.

A-II-1 Reverzy

Na polici je n pozícií a na každej z nich je položené jedno jabĺčko. Pozície sú očíslované zľava doprava od 0 po $n-1$. Jabĺčko na pozícii i má veľkosť a_i. Všetky jabĺčka majú navzájom rôzne veľkosti.
Chceli by sme všetky jabĺčka usporiadať podľa veľkosti, od najmenšieho po najväčšie. Kvôli hygienickým predpisom ich však nemôžeme presúvať sami, musíme na to použiť techniku.
V miestnosti s jabĺčkami máme mechanické rameno. Pomocou neho vieme zobrať jabĺčka z ľubovoľného súvislého úseku police a vrátiť ich naspäť na policu na tie isté miesta, avšak v presne opačnom poradí. Takejto operácii budeme hovoriť *reverz*. Každý reverz trvá rovnako dlho, bez ohľadu na to, aký dlhý úsek obraciame.

Súťažná úloha

Daný je popis police s jabĺčkami. Napíšte program, ktorý ich pomocou mechanického ramena čo najefektívnejšie usporiada.

Formát vstupu a výstupu

V prvom riadku vstupu je číslo n: počet jabĺčok.
V druhom riadku vstupu sú ich veľkosti: **navzájom rôzne** kladné celé čísla $a_0, \ldots, a_{n-1}$.
Na výstup vypíšte niekoľko riadkov: jeden pre každý reverz, ktorý chcete spraviť, v chronologickom poradí. Pre každý reverz vypíšte najmenšie a najväčšie číslo pozície v úseku, ktorý chcete reverznúť.

Hodnotenie

Primárnym kritériom hodnotenia vašich riešení je *rádový* počet reverzov, ktoré potrebujú spraviť.
(Na konštantách nezáleží, teda napr. n^3 reverzov je rovnako dobré ako $7n^3 + 40$ reverzov.)
Sekundárnym kritériom je časová zložitosť vášho programu.
Za riešenie, ktorému stačí lineárny počet reverzov ale má až kvadratickú časovú zložitosť, môžete získať 6 bodov.
Za riešenie, ktoré potrebuje väčší ako lineárny počet reverzov, môžete získať nanajvýš 4 body.

Príklady

vstup

```
7
10 50 40 30 70 60 20
```

výstup

```
4 6
1 4
```

Najskôr reverzneme úsek 4-6, čím dostaneme poradie jabĺčok 10 50 40 30 20 60 70, a potom reverzneme úsek 1-4, čím dostaneme usporiadané poradie.

vstup

```
7
20 30 40 50 60 70 10
```

výstup

```
5 6
4 5
3 4
2 3
1 2
0 1
```

Pre tento vstup existujú aj iné riešenia s menším počtom reverzov ako to ukázané v príklade výstupu.

A-II-2 Potrubie

Potrebujeme potrubie dĺžky **presne** ℓ centimetrov, ktoré bude mať **aspoň** f filtrov.
Máme niekoľko kusov rúry. Každý kus má svoju dĺžku d_i a svoj maximálny prietok za sekundu v_i. Niektoré kusy rúry majú v sebe filter, iné nie.
Hodnoty ℓ aj d_i sú rozumne malé **kladné celé čísla**.

Súťažná úloha

Zistite, či vieme z kusov rúry, ktoré máme, naskladať potrubie s požadovanými parametrami. Ak áno, zistite tiež, aký najväčší prietok môže toto potrubie mať. (Prietok potrubia je minimom z prietokov rúr, z ktorých sa skladá.)

Formát vstupu a výstupu

V prvom riadku vstupu sú čísla ℓ a f. V druhom riadku vstupu je číslo n: počet rúr. Zvyšok vstupu tvorí n riadkov. Každý z nich je tvaru „d_i v_i f_i“ a popisuje jednu rúru (f_i je `A` ak rúra filter obsahuje, resp. `N` ak nie).

Na výstup vypíšte maximálny prietok hľadaného potrubia, alebo -1 ak sa hľadané potrubie nedá postaviť.

Hodnotenie

Na plný počet bodov potrebujete riešenie, ktoré efektívne vyrieši vstupy s $n, \ell, f \leq 10\,000$.
Najviac 7 bodov môžete získať za riešenie, ktoré efektívne vyrieši vstupy s $n, \ell, f \leq 500$.
Najviac 5 bodov môžete získať za riešenie, ktoré efektívne vyrieši vstupy s $n, \ell \leq 500$ a $f = 0$.
Najviac 3 body môžete získať za riešenie, ktoré efektívne vyrieši vstupy s $n \leq 20$.

Príklady

vstup

```
100 0
4
40 1500 N
60 700 A
25 850 N
35 2700 A
```

výstup

```
850
```

Optimálne je použiť prvú, tretiu a štvrtú rúru. Prvá a druhá rúra tiež tvoria potrubie správnej dĺžky, avšak s menším prietokom.

vstup

```
200 2
3
100 1700 N
100 1500 A
100 1300 A
```

výstup

```
1300
```

Nevieme mať prietok väčší ako 1300, lebo musíme použiť obe rúry s filtrom.

vstup

```
100 0
1
110 4747 A
```

výstup

```
-1
```

Jediná rúra, čo máme, je pridlhá.

A-II-3 Vírus

Tajný agent James Dlhopis v prestrojení za inštalatéra úspešne infiltroval nepriateľskú ambasádu. Na USB kľúči si so sebou priniesol sofistikovaný vírus, ktorým by chcel nakaziť všetky počítače na ambasáde.
Na ambasáde je n počítačov. Niektoré dvojice počítačov momentálne vedia priamo komunikovať. Takéto dvojice nazveme susedné.
Bežné vírusy fungujú tak, že nakazený počítač postupne nakazí všetkých svojich susedov. Jamesov vírus je však sofistikovanejší. Aby sa ťažšie odhalilo, odkiaľ vlastne prišiel, šíri sa tak, že každý nakazený počítač postupne nakazí všetkých *susedov svojich susedov*.
Nakaziť počítač vírusom z USB kľúča trvá dosť dlho a James pri tom riskuje, že bude prichytený a odhalaný. Aby James minimalizoval riziko, chce takto nahrať vírus do počítača len raz.
Problém je v tom, že pri súčasnom stave prepojení medzi počítačmi sa môže stať, že sa vírus nedokáže rozšíriť na všetky ostatné počítače. James sa preto rozhodol, že skôr, ako nejaký počítač nakazí, niektoré dvojice počítačov ešte prepojí novými káblami. Aj toto by ale bolo dobré spraviť čo najrýchlejšie.

Súťažná úloha

Daný je popis súčasného stavu počítačovej siete na ambasáde. Napíšte program, ktorý vypočíta inštrukcie pre Jamesa: najskôr mu nájde jednu **najmenšiu možnú** sadu nových káblov, ktoré má natiahnuť, a potom mu povie číslo počítača, na ktorý má z USB kľúča nahrať vírus.

Formát vstupu a výstupu

V prvom riadku vstupu sú čísla $n \geq 3$ a m: počet počítačov na ambasáde a počet dvojíc susedných počítačov. Počítače majú čísla od 0 po $n - 1$. Zvyšok vstupu tvorí m riadkov. V každom z týchto riadkov sú čísla dvoch počítačov, ktoré susedia (t.j. už vedia priamo komunikovať medzi sebou).
Na výstup postupne vypíšte popis všetkých káblov, ktoré má James pridať, a na záver číslo počítača, na ktorý má nahrať vírus.

Hodnotenie

Plný počet bodov dostanú riešenia, ktoré efektívne vyriešia vstupy s $n \leq 100\,000$ a $m \leq 500\,000$.
Riešenia efektívne pre $n \leq 5000$ môžu získať maximálne 7 bodov.
Nezabudnite, že dôležitou súčasťou riešenia je dôkaz jeho správnosti.

Príklady

vstup

```
4 6
0 1
0 2
0 3
1 2
1 3
2 3
```

výstup

```
0
```

Netreba pridávať žiadne káble, stačí nakaziť počítač číslo 0. Ten potom priamo nakazí všetky ostatné počítače. (Napríklad počítač 1 je sused suseda počítača 0, keďže 0 susedí s 3 a 3 susedí s 1.)

vstup

```
4 2
0 2
2 3
```

výstup

```
3 0
0 1
1
```

Po tom, ako James natiahne dva nové káble, bude počítač 1 susediť s počítačom 0 a ten bude susediť s počítačmi 2 a 3. Keď teda James nakazí počítač 1, tento počítač nakazí počítače 2 a 3. Následne jeden z týchto počítačov nakazí počítač 0. (Počítač 0 je susedom suseda počítača 2, lebo 0 susedí s 3 a 3 susedí s 2.)

A-II-4 Exaktné exponenciálne algoritmy

Táto súťažná úloha nadväzuje na úlohu z domáceho kola. Za samotným zadaním úlohy nájdete študijný text. Ten je totožný so študijným textom zo zadaní domáceho kola. Podúlohy sú nezávislé, môžete ich riešiť v ľubovoľnom poradí.

Podúloha A (5 bodov).
Máme n povinností, ktoré už mali byť všetky hotové, a teda ich treba spraviť čo najskôr. Nedá sa plniť viac povinností naraz, treba ich plniť postupne jednu po druhej. Prácu na povinnostiach smieme ľubovoľne prerušovať a striedať (napr. 10 minút robiť na prvej, potom 5 minút na druhej, potom 2 minúty nič, a potom opäť 10 minút na prvej). Povinnosti sú očíslované od 1 po n. Na splnenie povinnosti i treba dokopy odpracovať t_i sekúnd.

Niektoré povinnosti je odporúčané dokončiť v správnom poradí. Presnejšie, pre každú dvojicu povinností i, j je zadaný nezáporný postih $s_{i,j}$, ktorý dostaneme, ak dokončíme skôr povinnosť i ako povinnosť j.

Navyše platí, že povinnosti treba mať splnené čo najskôr. Ak bude povinnosť i hotová po x sekundách od začiatku, budeme mať za ňu postih $p(i, x)$. Pre každú povinnosť veľkosť postihu v závislosti od času rastie, teda ak $x_1 < x_2$, tak $p(i, x_1) < p(i, x_2)$. Pre rôzne povinnosti môžu tomu istému času skončenia zodpovedať rôzne postihy. Konkrétny vzhľad funkcie $p(i, x)$ nepoznáte, môžete ju však vo svojom programe používať (ako keby ste mali k dispozícii knižnicu, ktorá túto funkciu počíta).

Chceme splniť všetky povinnosti tak, aby bol súčet postihov (za poradie dokončenia aj čas dokončenia dokopy) čo najmenší. Navrhnite algoritmus s časovou zložitosťou $\hat{O}(2^n)$, ktorý zistí, ako to vieme dosiahnuť. Nezabudnite na dôkaz správnosti.

Podúloha B (5 bodov).
Mesto je tvorené n lokalitami a m ulicami. Každá ulica je obojsmerná a spája niektoré dve lokality. V niektorých lokalitách by sme chceli postaviť búdky s verejnými záchodmi, a to tak, aby každá ulica mala aspoň na jednom konci záchod. Navrhnite čo najefektívnejší algoritmus, ktorý vypočíta najmenší počet búdok, ktoré stačí postaviť, ak vhodne zvolíme lokality pre ne.
Pri odhade časovej zložitosti nie je nutné vyčísliť konštantu – čas. zložitosť stačí uviesť napr. v tvare „$\hat{O}(c^n)$, kde c je hodnota spĺňajúca nasledujúci vzťah: . . . “.

Študijný text: Exaktné exponenciálne algoritmy

Pri analýze algoritmov sa často stretneme so zjednodušeným tvrdením, že algoritmy s *polynomiálnou* časovou zložitosťou považujeme za efektívne, zatiaľ čo algoritmy s *exponenciálnou* (a horšou) časovou zložitosťou považujeme za neefektívne. V tomto ročníku olympiády trochu nahliadneme do sveta exponenciálnych algoritmov a uvidíme, že toto zjednodušenie nemusí byť vždy pravdivé.

Porovnanie časových zložitostí

Pre súčasné počítače môžeme odhadnúť, že za minútu zvládnu vykonať približne 10^{10} jednoduchých logických krokov programu. Ak teda máme algoritmus s časovou zložitosťou $f(n)$ a zaujíma nás, aké veľké vstupy dokáže za minútu vyriešiť, hľadáme jednoducho najväčšie n také, že $f(n) \leq 10^{10}$. Výsledky pre niektoré zaujímavé funkcie uvádzame v tabuľke.

$f(n)$	$n \log n$	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

Vidíme teda, že napríklad medzi polynomiálnou časovou zložitosťou $3n^4$ a exponenciálnou časovou zložitosťou 1.1^n nie je v praxi zase až taký veľký rozdiel: rozsahy efektívne riešiteľných vstupov majú oba skoro rovnako veľké.

Možno vám v tabuľke padlo do oka, že 66 je dvakrát 33. Toto nie je náhoda. Totiž $\sqrt{2}^n = \left(2^{1/2}\right)^n = 2^{n/2}$. No a to znamená, že ak časovú zložitosť algoritmu zlepšíme z 2^n na $\sqrt{2}^n \approx 1.42^n$, tak nový algoritmus zvládne za rovnaký čas vyriešiť približne dvakrát väčší vstup ako ten pôvodný.
Túto úvahu vieme aj zovšeobecniť. Výraz a^n môžeme upraviť nasledovne: plati $a = 2^{\log_2 a}$, a teda $a^n = \left(2^{\log_2 a}\right)^n = 2^{n \log_2 a}$. Napríklad takto dostaneme, že 1.1^n je približne to isté ako $2^{0.1375n}$, resp. $2^{n/7.27254}$. Zlepšenie časovej zložitosti z 2^n na 1.1^n teda znamená, že novým algoritmom v rovnakom čase vyriešime vyše sedemkrát väčší vstup ako pôvodným. Všeobecne teda platí, že každé zlepšenie základu exponenciálnej funkcie niekoľko*krát* zväčší rozsah vstupov, ktoré ešte vieme efektívne riešiť.

O ťažkých problémoch

V teoretickej informatike poznáme značné množstvo algoritmických problémov, ktoré sú *ťažké*: nepoznáme pre ne žiadne algoritmy s polynomiálnou časovou zložitosťou a často máme dobré dôvody domnievať sa, že takáto časová zložitosť sa pre tieto problémy vôbec nedá dosiahnuť. (Exaktný dôkaz tejto domnienky pre jednu konkrétnu sadu ťažkých problémov je jedným z najvýznamnejších otvorených problémov v informatike.)
Z pozorovaní, ktoré sme spravili v predchádzajúcej časti študijného textu, však vyplýva jeden možný „smer útoku“: ak narazíme na takýto problém a potrebujeme ho vedieť exaktne riešiť, jednou z možností, ktoré máme, je snažiť sa nájsť taký exponenciálny algoritmus, ktorého základ exponenciálnej funkcie bude čo najmenší. Čím bližšie k jednotke ho dostaneme, tým väčšie vstupy ešte zvládneme v rozumnom čase vyriešiť.
Vo zvyšku tohto študijného textu si ukážeme dva takéto ťažké problémy a predvedieme si na nich dve techniky návrhu šikovných exponenciálnych algoritmov.

Zápis časovej zložitosti exponenciálnych algoritmov

Pri odhade časovej zložitosti klasických efektívnych algoritmov zvykneme zanedbávať konštanty. Namiesto exaktného vyčíslenia, že algoritmus na vstupe veľkosti n spraví nanajvýš $7n^2 - 3n + 147$ krokov, sa uspokojíme s asymptotickým odhadom „časová zložitosť algoritmu je $O(n^2)$“ – čiže „časová zložitosť je nejaká funkcia, ktorá rastie nanajvýš rádovo tak rýchlo ako funkcia n^2“.
Pri analýze exponenciálnych algoritmov niekedy budeme podobným spôsobom chcieť zanedbať aj polynomiálne faktory. Takýto horný odhad budeme zapisovať $\hat{O}$. Napríklad funkcie 1.9^n, $100 \cdot 2^n$ aj $(3n^2 + 6)2^n + n^4$ patria do $\hat{O}(2^n)$, ale funkcia $0.047 \cdot 2.01^n$ tam už nepatrí.
Formálne, funkcia f patrí do $\hat{O}(g)$ práve vtedy, ak patrí do $O(p \cdot g)$ pre nejaký polynóm p.

Časová zložitosť rekurzívnych programov

Niektoré exaktné exponenciálne algoritmy sú založené na *backtrackingu* (teda rekurzívnom prehľadávaní s návratom). Pri analýze ich časovej zložitosti budeme používať nasledujúce tvrdenie:

Veta o časovej zložitosti rekurzie. Majme rekurzívny algoritmus A, ktorý pri riešení problému postupuje nasledovne: Ak má vstup malej konštantnej veľkosti, vyrieši ho v konštantnom čase. Vo všeobecnom prípade pre vstup veľkosti n postupne spraví k rekurzívnych volaní, pričom pri i-tom z nich sa zavolá na vstup veľkosti nanajvýš $n - a_i$. (Hodnoty k aj a_i sú konštanty, ktoré ostávajú rovnaké počas celého behu algoritmu.) Okrem týchto rekurzívnych volaní už algoritmus spraví len polynomiálne veľa krokov v závislosti od n. Potom platí, že časová zložitosť tohto algoritmu je $\hat{O}(\alpha^n)$, kde α je jediné kladné reálne riešenie rovnice

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

Náčrt dôkazu. Keď si označíme časovú zložitosť nášho algoritmu T, z popisu algoritmu A dostávame, že T spĺňa rekurentný vzťah: $T(n) = T(n-a_1) + \cdots + T(n-a_k) + p(n)$, kde p je nejaký polynóm. Ak zanedbáme p a hľadáme čistú exponenciálnu funkciu T spĺňajúcu tento rekurentný vzťah, teda položíme $T(n) = \alpha^n$, dostaneme pre α práve vyššie uvedenú rovnicu. Následne sa dá ukázať, že keď za α zoberieme jej kladné reálne riešenie, tak náš algoritmus skutočne spraví $O(\alpha^n)$ rekurzívnych volaní, a teda celkový čas jeho behu vieme zhora odhadnúť $O(p(n) \cdot \alpha^n)$.

Príklady použitia. Ak algoritmus pri riešení problému veľkosti n spraví dve rekurzívne volania na problémy veľkosti $n-1$, dostávame rovnicu $x^n - x^{n-1} - x^{n-1} = 0$. Keďže hľadáme kladný reálny koreň, môžeme obe strany vydeliť nenulovým výrazom x^{n-1} a dostávame $x - 1 - 1 = 0$, čiže $x = 2$. Tento algoritmus má teda časovú zložitosť $\hat{O}(2^n)$.
Ak však algoritmus spraví jedno rekurzívne volanie na problém veľkosti $n-1$ a jedno na problém veľkosti $n-3$, dostaneme rovnakou úvahou rovnicu $x^3 - x^2 - 1 = 0$. Tej jediný kladný reálny koreň je $x \approx 1.4656$. Platí teda, že takýto algoritmus má časovú zložitosť $\hat{O}(1.4656^n)$. (Technický detail: všetky približné číselné konštanty uvádzame zaokrúhlené *nahor*.)

Maximálna nezávislá množina

V zoologickej záhrade práve postavili nový výbeh. Majú n zvierat, ktoré by doň chceli vypustiť. Problém je ale v tom, že niektoré dvojice zvierat nemôžu byť spolu vo výbehu, lebo by sa hrýzli. Na vstupe dostanete zoznam všetkých m takýchto dvojíc. Navrhnite algoritmus, ktorý zistí, koľko najviac zvierat môže skončiť vo výbehu.

Skôr, než sa pustíme do lepších riešení, podotknime, že túto úlohu vieme ľahko riešiť s časovou zložitosťou $O(m2^n)$: Existuje presne 2^n rôznych podmnožín zvierat. Postupne každú z nich vygenerujeme a zakaždým prejdeme celý zoznam dvojíc a pozeráme sa, či sme náhodou nevybrali obe zvieratá z niektorej dvojice.

Maximálna nezávislá množina: lepší algoritmus 1

Náš algoritmus bude mať podobu rekurzívnej funkcie, ktorá dostane na vstupe nejakú množinu zvierat a na výstupe vráti číslo hovoriace koľko najviac spomedzi týchto zvierat môžeme dať do prázdneho výbehu.
Všimnime si nejaké zviera z. Optimálne riešenie, ktoré **neobsahuje** z, nájdeme tak, že sa rekurzívne zavoláme na všetky zvieratá okrem z. Ako nájsť optimálne riešenie, ktoré **obsahuje** z? Označme $N(z)$ množinu tých zvierat, ktoré nemôžu byť vo výbehu spolu so zvieraťom z. Ak sa rozhodneme do výbehu pustiť zviera z, vieme, že zvieratá z $N(z)$ do výbehu pustiť nemôžeme. Optimálne riešenie obsahujúce z teda vieme získať tak, že sa rekurzívne zavoláme na všetky zvieratá okrem z a zvierat z $N(z)$, a následne do takto získaného riešenia pridáme zviera z. Na výstup naša funkcia vráti väčšie z oboch vyššie popísaných riešení.
Je zjavné, že za zviera z sa oplatí voliť také zviera, ktoré má *čo najviac* konfliktov s inými – aby sme pri druhom rekurzívnom volaní dostali čo najmenšiu množinu zvyšných zvierat. Toto pozorovanie nás privádza k nasledujúcemu algoritmu:

1. Ak má každé zvieratko najviac jeden konflikt: Zober všetky bezkonfliktné zvieratká. Z každej dvojice, čo je v konflikte, zober ľubovoľné jedno. Hotovo.
2. Inak si vyber zvieratko z ktoré má najviac konfliktov s inými.
3. Rekurzívne nájdi najlepšie riešenie pre všetky zvieratká okrem z.
4. Rekurzívne nájdi najlepšie riešenie pre všetky zvieratká okrem z a $N(z)$, a pridaj doň z.
5. Na výstup vráť lepšie z týchto dvoch riešení.

Pre veľké vstupy tento algoritmus vždy spraví dve rekurzívne volania. Prvé je na vstup veľkosti $n-1$. Keďže vybrané zvieratko z má aspoň dva konflikty, druhé rekurzívne volanie je na problém veľkosti nanajvýš $n-3$. Z vety o časovej zložitosti rekurzie teda vyplýva, že toto riešenie má časovú zložitosť $\hat{O}(1.4656^n)$.

Maximálna nezávislá množina: lepší algoritmus 2

Aj tento algoritmus bude mať podobu rekurzívnej funkcie, ktorá dostane na vstupe nejakú množinu zvierat a na výstupe vráti číslo hovoriace koľko najviac spomedzi týchto zvierat môžeme dať do prázdneho výbehu.
Pripomeňme si, že optimálne riešenie, ktoré **obsahuje** zviera z, vieme nájsť tak, že nájdeme optimálne riešenie pre všetky zvieratá okrem z a $N(z)$, a potom doň pridáme z. Tentokrát budeme pokračovať trochu inou myšlienkou. Tvrdíme, že v optimálnom riešení, ktoré z **neobsahuje**, musí byť vo výbehu aspoň jedno zo zvierat v $N(z)$. Toto je dosť zjavné: riešenie, v ktorom nie je vo výbehu ani z ani žiadne zviera z $N(z)$, nemôže byť optimálne, lebo ho vieme zlepšiť pustením z do výbehu.
Uvažujme teda nasledujúci algoritmus (slovo „najmenej“ v kroku 1 vysvetlíme nižšie):

1. Vyber si zvieratko z, ktoré má **najmenej** konfliktov s inými.
2. Pre každé zvieratko y z množiny $\{z\} \cup N(z)$:
   – Rekurzívnym volaním nájdi najlepšie riešenie pre všetky zvieratká okrem y a $N(y)$.
   – Pridaj doň y, čím dostaneš najlepšie riešenie obsahujúce y.
3. Na výstup vráť najlepšie z riešení zostrojených v predchádzajúcom kroku.

Príklad. Nech z je zebra a nech naraz s ňou nemôže byť vo výbehu kôň, srnka ani antilopa. Potom v optimálnom riešení je aspoň jedno z týchto štyroch zvierat. Postupne teda pre každé z nich nájdeme rekurzívnym volaním najlepšie riešenie, ktoré ho obsahuje.

Nech má vybrané zvieratko k konfliktov. Z toho, ako sme zvolili zvieratko z v kroku 1, vyplýva, že **každé** zvieratko má aspoň k konfliktov. Potom tento algoritmus spraví $k+1$ rekurzívnych volaní, pričom každé z nich bude na nejaký nový problém s nanajvýš $n-(k+1)$ zvieratkami.
Dá sa ukázať, že najhorší prípad nastáva pre $k=2$, teda keď má každé zvieratko presne dva konflikty. Vtedy bude mať tento algoritmus časovú zložitosť $\hat{O}(3^{n/3})$, čo vieme upraviť do podoby $\hat{O}(1.4423^n)$.

Maximálna nezávislá množina: nájdenie všetkých optimálnych riešení

„Lepší algoritmus 2“, ktorý sme si práve popísali, vieme ľahko upraviť tak, aby nie len vypočítal najväčší počet zvierat vo výbehu, ale navyše aj postupne vygeneroval a vypísal všetky **optimálne** riešenia tejto úlohy. Z toho vyplýva, že optimálnych riešení nemôže byť viac ako $\hat{O}(3^{n/3})$. Je ich teda vždy výrazne menej ako 2^n.
Ľahko nahliadneme, že tento odhad je pomerne tesný. Stačí zobrať $n=3k$ zvieratiek, rozdeliť ich do trojíc a povedať, že v každej trojici sú každé dve zvieratá v konflikte. Potom bude každé optimálne riešenie obsahovať práve jedno zviera z každej trojice a teda bude presne $3^k = 3^{n/3}$ optimálnych riešení.

Problém obchodného cestujúceho

V krajine je n miest, očíslovaných od 1 po n. Pre každú dvojicu miest (i,j) poznáme cenu $c_{i,j}$ cestovného lístku z i do j. Obchodný cestujúci Emil potrebuje precestovať celú krajinu: chce začať v meste 1, postupne navštíviť **práve raz** každé iné mesto a nakoniec sa vrátiť späť do mesta 1. Koľko peňazí mu na to stačí?

Priamočiare riešenie tejto úlohy má časovú zložitosť ešte horšiu ako exponenciálnu. Emila zaujíma, v akom poradí má navštíviť mestá 2 až n, hľadá teda ich optimálnu *permutáciu*. Toto vieme spraviť tak, že postupne vygenerujeme všetkých $(n-1)!$ permutácií miest 2 až n a pre každú spočítame, koľko by nás stálo cestovné. Takéto riešenie má teda časovú zložitosť $\hat{O}(n!)$.

Problém obchodného cestujúceho: dynamické programovanie

Ukážeme si, ako túto úlohu vyriešiť s časovou zložitosťou $\hat{O}(2^n)$, presnejšie, v čase $O(n^2 2^n)$.
Všimnime si Emila niekedy počas jeho cesty po krajine. Už navštívil nejaké mestá a zaplatil nejaké peniaze za cestovné. Položme mu teraz otázku: „Za koľko najmenej peňazí vieš svoju cestu dokončiť?“
Od čoho závisí odpoveď na túto otázku? Len od dvoch vecí: od mesta a, kde sa Emil práve nachádza, a od množiny miest B, ktoré ešte nenavštívil. Označme $d_{a,B}$ odpoveď na otázku s týmito dvoma parametrami.
Ak je množina B prázdna, je otázku ľahké zodpovedať: $d_{a,\emptyset} = c_{a,1}$, lebo už sa len potrebujeme vrátiť z aktuálneho mesta a na začiatok. Vo všetkých ostatných prípadoch sa pozrime na to, čo Emil spraví v nasledujúcom kroku: vyberie si nejaké mesto $b \in B$ a odcestuje doň. Najlepšie riešenie pre konkrétne mesto b bude Emila stáť $c_{a,b} + d_{b,B-\{b\}}$ peňazí: najskôr zaplatí $c_{a,b}$ za cestu z a do b a potom $d_{b,B-\{b\}}$ za optimálne dokončenie riešenia zo situácie, kedy stojí v meste b a ešte potrebuje navštíviť ostatné mestá z množiny B.
Hodnotu $d_{a,B}$ pre $B \neq \emptyset$ teda spočítame tak, že postupne vyskúšame všetky $b \in B$, pre každé z nich zistíme, k ako najlepšiemu riešeniu vedie, a z takto získaných hodnôt zoberieme minimum.
Otázok, ktoré si kladieme, teda rôznych hodnôt $d_{a,B}$, ktoré nás zaujímajú, je $O(n2^n)$, lebo je n možností pre a a pri konkrétnom a nanajvýš 2^{n-1} možností pre B (lebo pre každé mesto iné od a máme dve možnosti: buď v B leží alebo nie). Každú otázku vieme zodpovedať v čase $O(n)$, a teda celková časová zložitosť výpočtu všetkých hodnôt $d_{a,B}$ je $O(n^2 2^n)$. Výsledným riešením je potom hodnota $d_{1,\{2,3,\ldots,n\}}$.

Pseudokód rekurzívnej implementácie:

```
funkcia d(a, B):
    ak si už niekedy spracúval vstup (a, B):
        vráť zapamätanú odpoveď

    ak je B prázdna:
        odpoveď = C[a, 1]
    inak:
        odpoveď = minimum { C[a, b] + d(b, B okrem b)  |  b je prvok B }
    zapamätaj si že pre vstup (a, B) je výstup odpoveď
    vráť odpoveď
```

Všimnite si, že pri každom rekurzívnom volaní sa zmenší množina ešte nenavštívených miest. Vďaka tomu každá vetva rekurzie eventuálne skončí. Pre každú z $O(n2^n)$ dvojíc (a, B) sa telo tejto funkcie (výpočet konkrétnej hodnoty $d_{a,B}$) vykoná nanajvýš raz, vďaka čomu dosiahneme sľúbenú celkovú časovú zložitosť $O(n^2 2^n)$.

Pseudokód jednej možnej iteratívnej implementácie:

```
pre každé a:
    D[a, prázdna množina] = C[a, 1]

pre každú veľkosť vb množiny B od 1 až po n-1:
    pre každú množinu B veľkosti vb:
        pre každé a nepatriace do množiny B:
            D[a, B] = nekonečno
            pre každé b z množiny B:
                D[a, B] = min( D[a, B], C[a, b] + D[b, B okrem b] )
```

Všimnite si, že pri tejto implementácii pri výpočte konkrétnej $d_{a,B}$ už poznáme všetky hodnoty $d_{b,B-\{b\}}$, ktoré potrebujeme, lebo sme ich vypočítali v skoršej iterácii vonkajšieho for-cyklu: množina $B-\{b\}$ má menšiu veľkosť ako množina B.

Na záver tohto študijného textu podotkneme, že pri praktickej implementácii tohto algoritmu by sme na uloženie množín B použili tzv. bitové masky (bitmasky): množinu $\{x_1, \ldots, x_i\}$ by sme reprezentovali číslom $2^{x_1} + \cdots + 2^{x_i}$, teda číslom, ktoré má nastavené práve bity s číslami $x_1, \ldots, x_i$.
Rozmyslite si, že ak má konkrétna množina priradené nejaké číslo, tak všetky jej podmnožiny majú priradené menšie čísla (lebo keď zmažeme z množiny prvok, tak v dvojkovom zápise čísla, ktoré ju reprezentuje, zmeníme príslušnú jedničku na nulu). Namiesto vonkajších dvoch for-cyklov by sme teda mohli použiť len jeden for-cyklus cez všetky čísla predstavujúce platné kódy množín, od najmenšieho po najväčšie. Takto dostaneme iné poradie, v ktorom budeme množiny B spracúvať, ale vyššie popísaný algoritmus bude stále korektne fungovať, lebo pre každé B a b bude aj teraz platiť, že množinu $B - \{b\}$ spracujeme skôr ako množinu B.

TRIDSIATY PIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2020