Korešpondenčný seminár z programovania
XXXIV. ročník, 2016/17
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Návody k úlohám 2. kola zimnej časti kategórie T

V kategórii T neuvádzame vzorové riešenia ale skôr návody na riešenie úloh. Ich cieľom je prezradiť vám hlavnú myšlienku riešenia, aby ste podľa návodu mohli riešenie domyslieť sami. Občas teda vynecháme niektoré drobné detaily, neuvádzame implementácie dátových štruktúr a nerozpisujeme niektoré kroky.

Neuvádzame ani vzorové programy, pretože chceme, aby ste po prečítaní návodu naprogramovali tie úlohy, ktoré ste nespravili počas trvania série. Keď si tieto riešenia sami naprogramujete, naučíte sa tým oveľa viac ako pozeraním sa na zdrojové kódy.

Implementácie všeobecne známych algoritmov a dátových štruktúr môžete nájsť vo vzorových riešeniach kategórií Z a O starších ročníkov KSP alebo aj na internete.

1. Takto sa obsadzuje graf

návod písal Žaba
(max. 0 b za popis, 20 b za program)

Ako vždy, začnime sériou pozorovaní. Uvedomme si, že naším cieľom je **iba dobyť všetky vrcholy** a teda nemusíme dobyť všetky hrany. Dokonca hrany, ktoré dobyjeme ani nemusia vytvárať kostru nášho grafu. Ak si preto zoberieme nejaké optimálne výsledné riešenie, tak dobyté hrany budú vytvárať niekoľko samostatných komponentov. Vieme teraz vypočítať koľko nás takéto dobytie stálo?

Samozrejme, komponenty sa nijak neovplyvňujú, keďže figúrky medzi nimi nemôžu prechádzať. Každý teda musíme započítať zvlášť. Takisto je jasné, že v jednom komponente chceme figúrky pridávať iba do jedného vrcholu. Toho, v ktorom je to najlacnejšie. Cenu pridania jednej figúrky do vrcholu v nám určuje hodnota b_v. Preto chceme nájsť v danom komponente vrchol s najmenšou hodnotou b_v. Označme si túto hodnotu premennou *cena*.

A koľko figúrok potrebujeme na to, aby sme dobyli celý tento komponent? Určite potrebujeme dobyť každý vrchol. Na dobytie vrchola v potrebujeme a_v figúrok. Zo všetkých vrcholov v tomto komponente preto nájdeme ten, na ktorého dobytie treba najviac figúrok, teda má najväčšiu hodnotu a_v, a túto hodnotu si označíme premennou $pocet_v$. Je jasné, že na dobytie všetkých vrcholov potrebujeme aspoň $pocet_v$ figúrok.

Ani takýto počet však nemusí postačovať, mohli sme sa totiž rozhodnúť v tomto komponente dobyť hranu e, na ktorej dobytie potrebujeme c_e figúrok, pričom $c_e > pocet_v$. V tomto komponente sa teda musíme pozrieť aj na **všetky dobyté hrany** a nájsť tú, ktorá má najväčšiu hodnotu c_e. Túto hodnotu si označíme ako $pocet_e$. Všimnite si, že v tomto komponente môžu byť aj drahšie hrany, ktoré sme sa však rozhodli nedobyť a preto nás nemusia zaujímať.

Teraz si rozmyslite, že ak naozaj vytvoríme $\max(pocet_v, pocet_e)$ figúrok vo vrchole, v ktorom to stojí *cena* peňazí, podarí sa nám potom dobyť celý tento komponent, zaplatíme za to $cena \cdot \max(pocet_v, pocet_e)$ peňazí a táto hodnota je najmenšia možná.

Ďalšie dobré pozorovanie je, že sa nám nikdy neoplatí dobýjať hranu z oboch strán – teda tak, že na oboch jej koncoch je nenulový počet figúrok. Ak by sme totiž na jednom konci mali x figúrok a na druhom y, tak sa radšej pozrieme, do ktorých vrcholov sme pôvodne pridali tieto figúrky, vyberieme si ten lacnejší a pridáme doňho $x + y$ figúrok. Uvedomme si, že stále budeme vedieť dobyť tú istú časť grafu a určite to nebude drahšie.

No a nakoniec, povedzme si, že chceme dobyť nejakú hranu e, na ktorej dobytie treba c_e figúrok. Ak si v grafe zoberieme iba hrany, ktorých dobytie je nanajvýš také drahé, tak sa nám opäť vytvorí viacero súvislých komponentov. Ak však máme dostatok figúrok na dobytie hrany e, určite budeme mať dosť figúrok na dobytie každej hrany, ktorá bola nanajvýš taká drahá a je v rovnakom komponente ako hrana e. Ak nás to teda nebude stáť nič naviac, tak sa nám tieto hrany určite oplatí dobyť. Vďaka nim sa totiž vieme dostať k viacerím vrcholom, čo nám dáva šancu nájsť vrchol s nízkou hodnotou b_v, v ktorom tých c_e figúrok ktoré potrebujeme, vieme vyrobiť.

Myšlienka a implementácia

Nepripomína vám to niečo? Komponenty, hrany lacnejšie ako nejaká hrana e? Lebo mne to smrdí Kruskalom – algoritmom na hľadanie najlecnejšej kostry. Aj ohľadom zložitosti tento algoritmus sedí, takže to vyzerá, že sme na dobrej ceste.

Kruskalov algoritmus začína tým, že si usporiada hrany od najlacnejšej po najdrahšiu a potom ich postupne pridáva do grafu. Pritom si, najčastejšie pomocou union-findu, udržiava množinu súvislých komponentov. Pri

pridávaní hrany najskôr skontroluje, či daná hrana nevedie medzi dvoma vrcholmi, ktoré už patria tomu istému komponentu. Ak áno, takúto hranu ignoruje. V opačnom prípade ju však do grafu vloží a spojí komponenty, do ktorých patrili koncové vrcholy tejto hrany.

Ako tento algoritmus upravíme tak, aby riešil náš problém? Zjavne najdôležitejšia je operácia pridávania hrany. Už teda existujú nejaké súvislé komponenty a predpokladajme, že o každom z nich vieme, ako čo najlacnejšie dobyť všetky vrcholy v ňom. Čo sa zmení pridaním hrany e?

Hrana e môže spájať vrcholy, ktoré patria do toho istého komponentu. Tak ako v klasickom Kruskalovi však túto hranu môžeme ignorovať. Ak už vieme ako najlacnejšie dobyť všetky vrcholy v danom komponente, pridanie hrany, na ktorej dobytie potrebujeme viac figúrok ako na zvyšné hrany v tomto komponente, nám určite riešenie nezlepší.

Takže sa zamerajme na prípad, keď hrana e vedie medzi dvoma komponentami. Ak sa rozhodneme túto hranu dobyť, bude platiť naše posledné pozorovanie. Teda následne budeme vedieť zadarmo dobyť aj zvyšné hrany v týchto dvoch komponentoch, čím nám vznikne jeden súvislý komponent. A koľko stojí dobytie tohto spoločného komponentu? Podľa prvého pozorovania musíme nájsť hodnoty *cena*, $pocet_v$ a $pocet_e$. Zjavne $pocet_e = c_e$, keďže aktuálne pridávaná hrana je drahšia ako všetky doteraz pridané hrany. Ostáva teda nájsť vrchol, na ktorého dobytie potrebujeme najviac figúrok a vrchol, v ktorom je pridávanie figúrok najlacnejšie.

Aby sme tieto dve hodnoty vedeli vypočítať rýchlo, budeme si pre každý komponent naviac pamätať aj jeho hodnotu *cena* a $pocet_v$. Pri spojení dvoch komponentov potom stačí zobrať minimum z ich hodnôt *cena* a maximum z ich hodnôt $pocet_v$. Vďaka tomu dokážeme v konštantnom čase vypočítať, koľko nás stojí dobytie tohto komponentu.

Občas sa nám však hranu e nemusí oplatiť dobyť. Napríklad preto, lebo má príliš veľkú hodnotu c_e. Môžeme sa však tváriť, že sa tieto dva komponenty spojili aj tak. Stále vieme vypočítať, za koľko najmenej dobyť všetky vrcholy tohto komponentu. Jednoducho si zoberieme súčet dobytí jednotlivých komponentov. A do budúcna nám to nič nepokazí, lebo ak sa niekedy v budúcnosti rozhodneme dobyť hranu, ktorá bude zasahovať do tohto komponentu tak budeme vedieť zadarmo dobyť aj hranu e (lebo je lacnejšia) a teda sa tváriť, že vlastne celý čas boli spolu. (Toto je asi najpodstatnejšia a teda aj najťažšia časť myšlienky tohto riešenie. Odporúčam prečítať ešte raz a poriadne požuť.)

Samotný algoritmus je potom naozaj len veľmi jemnou obmenou Kruskalovho algoritmu. Pre každý komponent si budeme pamätať tri hodnoty – *cena* (ako najlacnejšie viem vyrábať figúrky), $pocet_v$ (koľko najviac figúrok potrebujem na dobytie nejakého vrcholu v tomto komponente) a *riesenie* (najlacnejšia cena dobytia všetkých vrcholov tohto komponentu). Na začiatku tvorí každý vrchol samostatný komponent a spomenuté tri hodnoty sú nastavené na b_v, a_v a $b_v \cdot a_v$.

Následne budeme spracovávať hrany od tej s najmenším c_e po tú s najväčším. Pre každú hranu sa pozrieme, či spája vrcholy z rôznych komponentov. Ak nie, tak túto hranu ignorujeme. Inak si vypočítame, za akú cenu vieme dobyť všetky vrcholy v tomto komponente ak dobyjeme hranu e ($\min(cena, cena') \cdot \max(c_e, pocet_v, pocet'_v)$) alebo ak ju nedobyjeme ($riesenie + riesenie'$). Vyberieme si tú lacnejšiu možnosť, komponenty spojíme dokopy a príslušne upravíme hodnoty *cena*, $pocet_v$ a *riesenie* pre tento spojený komponent.

Na konci nám ostane jeden komponent a jeho hodnota *riesenie* nám hovorí, ako najlacnejšie dobyť všetky vrcholy v celom grafe.

návod písal Baklažán

2. Trápny vianočný stromček

(max. 0 b za popis, 20 b za program)

Táto úloha sa dala riešiť rôznymi spôsobmi, tu si ukážeme len jeden z nich.

Začnime s tým, že si náš strom zakoreníme (napríklad vo vrchole 1). Koreň nášho stromu označme K. Nech $\mathrm{Ans}(W, X, Y, Z)$ označuje odpoveď na otázku pre cesty z W do X a z Y do Z. Nech $\mathrm{Lca}(X, Y)$ označuje najnižšieho spoločného predka vrcholov X, Y a nech $P(X)$ označuje priameho predka vrcholu X. Potom vieme otázku $\mathrm{Ans}(W, X, Y, Z)$ rozbiť nasledovne:

$$\mathrm{Ans}(W, X, Y, Z) = \mathrm{Ans}(W, K, Y, Z) + \mathrm{Ans}(K, X, Y, Z) - \mathrm{Ans}(\mathrm{Lca}(W, X), K, Y, Z) - \mathrm{Ans}(K, \mathrm{P}(\mathrm{Lca}(W, X)), Y, Z).$$

Keď rovnakú inklúziu a exklúziu aplikujeme aj na cestu z Y do Z, rozbijeme našu pôvodnú otázku na 16 iných, pre ktoré však bude platiť, že obe cesty majú jeden koniec v koreni. Takýmto spôsobom si teda rozbijeme všetky otázky zo vstupu. Teraz nám stačí už len vyriešiť trochu jednoduchšiu verziu problému, kde v každej otázke obe cesty začínajú v koreni.

Namiesto toho, aby sme postupne spracovávali jednotlivé otázky (ľudí pozerajúcich sa na stromček) a počítali k nim odpovede, budeme postupne spracovávať jednotlivé druhy ozdôb a počítať, ako prispievajú vrcholy s

daným druhom ozdoby do odpovedí jednotlivých otázok. Keď spracujeme všetky druhy ozdôb, budeme zároveň vedieť odpovede na jednotlivé otázky.

Čísla (druhy ozdôb), ktoré sa vyskytujú na stromčeku, rozdeľme do dvoch skupín:

- *Časté* : čísla, ktoré sa v strome vyskytujú aspoň $\sqrt{n/\log n}$-krát[1].
- *Zriedkavé* : čisla, ktoré sa na strome vyskytujú menej ako $\sqrt{n/\log n}$ krát.

S každou skupinou čísel sa vysporiadame inak.

Časté čísla

Keďže každé časté číslo sa v strome vyskytuje veľakrát, týchto čísel bude málo. Preto môžeme na každom z nich stráviť pomerne veľa času. Pre každé časté číslo c teda môžeme spraviť na našom strome jedno DFS, v ktorom si pre každý vrchol vypočítame, koľkokrát sa vyskytuje číslo c na ceste z tohto vrchola do koreňa. Následne už pre každú otázku vieme vypočítať, ako do jej odpovede prispeje c: v konštantnom čase zistíme, koľko c-čok leží na jednej, a koľko na druhej ceste, a tieto dve čísla nám stačí vynásobiť. Nakoniec ešte bude treba odčítať počet c-čok, ktoré ležali na prieniku ciest (lebo dvojice obsahujúce ten istý vrchol dvakrát podľa zadania nemáme počítať).

Každé časté číslo takto vieme spravocať v čase $O(n+q)$. Keďže častých čísel bude najviac $n/\sqrt{n/\log n} = \sqrt{n\log n}$, spracovanie všetkých častých čísel bude dokopy trvať $O((n+q)\sqrt{n\log n})$.

Zriedkavé čísla

Najprv si prejdeme celým stromom a pre každé zriedkavé číslo si poznačíme, kde všade (v ktorých vrcholoch) sa toto číslo vyskytuje.

Vrcholy nášho (zakoreneného) stromu si môžeme očíslovať v preorder poradí (teda v poradí, v akom ich navštívi DFS). Toto poradie má jednu peknú vlastnosť: pre každý podstrom platí, že čísla vrcholov tohto podstromu tvoria súvislú postupnosť. Inými slovami, sú to všetky celé čísla z nejakého intervalu.

Každá otázka je daná štyroma vrcholmi: koncami ciest, na ktoré sa táto otázka pýta. Keďže však dva z týchto vrcholov sú vždy koreň nášho stromu, reálne je každá otázka určená iba dvoma vrcholmi. Ak sa na čísla týchto vrcholov (v našom novom preorder číslovaní) budeme pozerať ako na súradnice, na každú otázku sa môžeme pozerať ako na bod v dvojrozmernom priestore.

Nech c je teraz nejaké zriedkavé číslo. Pre každú dvojicu vrcholov, ktoré v sebe majú ozdobu typu c, sa pozrieme, ktoré otázky ovplyvnia. Nech U a V je nejaká dvojica vrcholov typu c. Táto dvojica prispieva do odpovedí práve tých otázok, ktorých jedna cesta končí v podstrome pod U a druhá v podstrome pod V. To znamená, že koniec jednej cesty musí mať číslo z intervalu zodpovedajúceho podstromu U a koniec druhej musí mať číslo z intervalu zodpovedajúceho podstromu V. V našej geometrickej predstave tomu zodpovedajú body ležiace vnútri nejakého obdĺžnika. Všetkým otázkam z tohto obdĺžnika by sme teda chceli zvýšiť odpoveď o 1. To však neurobíme hneď. Namiesto toho si iba zapamätáme súradnice tohto obdĺžnika.

Takto prejdeme všetky dvojice vrcholov, obsahujúcich rovnaké zriedkavé číslo a pre každú z nich dostaneme jeden obdĺžnik. Týchto obdĺžnikov bude dokopy najviac $n\sqrt{n/\log n}$, pretože každý vrchol bude v najviac $\sqrt{n/\log n}$ rôznych dvojiciach (keďže inak by už jeho číslo muselo byť časté a nie zriedkavé).

Máme teda zhruba $n\sqrt{n/\log n}$ obdĺžnikov a q bodov a pre každý bod nás zaujíma, v koľkých obdĺžnikoch leží. Všetky súradnice sú pritom celé čísla z rozsahu 1 až n. Takáto úloha sa dá vyriešiť zametaním za pomoci intervalového stromu (treba strom s lazy propagation), v čase $O((q+n\sqrt{n/\log n})\log n) = O(q\log n + n\sqrt{n\log n})$.

Celá úloha sa dá teda vyriešiť v čase $O((n+q)\sqrt{n\log n})$.

návod písal Hodobox

3. Turnaj Mladých Šachistov

(max. 0 b za popis, 20 b za program)

Táto úloha bola trochu neklasická v tom, že jej obtiažnosť nespočívala vo vymyslení riešenia, ale jeho implementácií. Vzorák sa totiž prakticky nachádzal v samotnom zadaní:

"{farba} hrac ma sach.", kde {farba} je "Biely", resp. "Cierny", ak je kráľ tohto hráča v ohrození, ale existuje platný ťah niektorou z jeho figúrok taký, po ktorom už v ohrození nebude.

Ak zistíme, že práve jeden kráľ je v ohrození, stačí nám vyskúšať každý ťah tohto hráča, a zakaždým sa spýtať znovu tú istú otázku – je kráľ v ohrození? Ak je aspoň raz odpoveď "nie", odpovedáme šach, inak mat.

[1] Ak vás zaujíma, ako sme prišli k číslu $\sqrt{n/\log n}$: toto číslo sme zvolili tak, aby bola výsledná časová zložitosť nášho algoritmu čo najmenšia. Ak by sme namiesto tejto hranice použili hodnotu $\sqrt{n}$, nemalo by to v praxi urobiť veľký rozdiel.

Vzhľadom na to že hlavným cieľom úlohy je donútiť riešiteľa si dopredu premyslieť ako túto jednu vetu naprogramovať čo najprehľadnejšie a bezbolestnejšie, hlavným cieľom vzoráku bude priblížiť ako k takémuto problému pristupovať, namiesto toho aby sa po vyslovení jednovetového riešenia prilepili strany hotového kódu.

Vzorové riešenie by teda, po prepísaní do pseudokódu, vyzeralo zatiaľ zhruba takto

```
string vyries()
{
    nacitaj a spracuj vstup

    B = je_ohrozeny(biely_kral,sachovnica)
    C = je_ohrozeny(cierny_kral,sachovnica)

    if B and C:
        return "Nemozna situacia."
    if !B and !C:
        return "Neutralna situacia."

    for priatelska_figurka in sachovnica:
        for nova_sachovnica in skus_vsetky_tahy(priatelska_figurka,sachovnica):
            if !je_ohrozeny(ohrozeny kral,nova_sachovnica):
                return farba + " hrac ma sach."

    return farba + " hrac ma mat."
}

main()
{
    t = input()
    while(t--)
        print(vyries())
}
```

Samozrejme vynechávam detaily ako načítanie vstupu a zapamätanie si pozícií kráľov a pod. Štruktúra je tá dôležitá časť. Toto riešenie by teda bolo kompletné, ak by boli hotové funkcie `je_ohrozeny(kto,kde)` , ktorá zistí či figúrka `kto` na šachovnici `kde` je ohrozená nepriateľskými figúrkami, a `skus_vsetky_tahy(kto,kde)` ktorá zoberie figúrku `kto` a šachovnicu `kde`, a vygeneruje všetky možné šachovnice ktoré vzniknú tým, že touto figúrkou spravíme jeden ťah.

A teraz sa zamyslite, či sú si tieto funkcie navzájom podobné. Čo to vlastne znamená, že figúrka je v ohrození? No predsa to, že nejaká nepriateľská figúrka má taký ťah, ktorým sa posunie na pozíciu našej figúrky. Inak povedané, na overenie ohrozenosti vlastne chceme vyskúšať všetky ťahy nepriateľských figúrok. `je_ohrozeny(kto,kde)` teda, opäť odhliadnuc od detailov, vyzerá takto

```
bool je_ohrozeny(kto,kde)
{
    for nepriatel in nepriatelske_figurky:
        for nova_sachovnica in skus_vsetky_tahy(nepriatel,kde):
            if nova_sachovnica[pozicia kto] == nepriatel:
                return true
    return false
}
```

Ostáva nám teda už len `skus_vsetky_tahy`. Máme vlastne dva typy figúrok: tie ktoré majú určitý počet pohybov (kôň a kráľ) a tie ktoré maju určitý počet smerov, v ktorých môžu ísť kým nenarazia na prekážku (okraj šachovnice alebo figúrku). Môžeme teda napríklad zostrojiť dve funkcie, jedna ktorá vyskúša dané pohyby (a použijeme ju pre kráľa a kone), a jednu ktorá vyskúša pohyby v daných smeroch. Pešiakov bude jednoduchšie naprogramovať osobitne. Celé to môže vyzerať zhruba nasledovne

```
sachovnica[] niekolko_pohybov(figurka kto,sachovnica kde)
{
    sachovnica[] results

    for pohyb in kto.pohyby:
        if mozem(kto.pozicia + pohyb):
            tmp = kde[kto.pozicia + pohyb]
            kde[kto.pozicia] = '.'
            kde[kto.pozicia + pohyb] = kto
            results.append(kde)
            kde[kto.pozicia] = kto
            kde[kto.pozicia+pohyb] = tmp

    return results
}

sachovnica[] pohyb_v_smere(figurka kto,sachovnica kde)
{
    sachovnica[] results

    for smer in kto.pohyby:
        pohyb = smer
        while mozem(kto.pozicia+pohyb):
            tmp = kde[kto.pozicia + pohyb]
            kde[kto.pozicia] = '.'
            kde[kto.pozicia + pohyb] = kto
```

```
            results.append(kde)
            kde[kto.pozicia] = kto
            kde[kto.pozicia+pohyb] = tmp
            pohyb += smer

    return results
}

sachovnica[] skus_vsetky_pohyby(figurka kto, sachovnica kde)
{
    if kto == pesiak:
        //riešiť samostatne (zvoliť správny smer podľa farby, útočiť šikmo...)

    if kto == kral or kto == kon:
        return niekolko_pohybov(kto,kde)

    return pohyb_v_smere(kto,kde)

}
```

Pritom `mozem(pozicia)` vráti `true` ak je naša figúrka povolená pohnúť sa na dané políčko (teda nie je mimo šachovnice a nie je na nej priateľská figúrka). Tú tu uvádzať hádam nemusím.

Toto je teda kompletná štruktúra riešenia. Samozrejme, ešte chýba kopa práce utlačiť syntaktické detaily – napríklad `pozicia` by asi mala byť dvojica súradníc, každej figúrke musíme priradiť množinu pohybov vo forme dvojíc {`dx,dy`}, čo a ako si pamätáme o šachovnici, a tak ďalej, skúsenejším by však táto časť už nemala robiť veľké problémy.

Ak by ste si mali z tejto úlohy niečo odniesť, tak je to to, že riešenie úlohy nie vždy končí vymyslením algoritmu, naopak, niekedy to len začína. Vtedy je dobré si všetko pomaly dopredu premyslieť – chvíľkou rozvážneho plánovania štruktúry vášho programo si často môžete ušetriť hodiny programovania a najmä debugovania, jednoduchej myšlienky komplikovaným či zdĺhavým spôsobom.