Korešpondenčný seminár z programovania
XXXII. ročník, 2014/15
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Návody k úlohám 1. kola letnej časti kategórie T

V kategórii T obvykle 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.

Výjnimkou je prvá úloha, ktorá je zároveň poslednou úlohou v kategórii O, pri ktorej obvykle použijeme ten istý vzorák ako v Óčku.

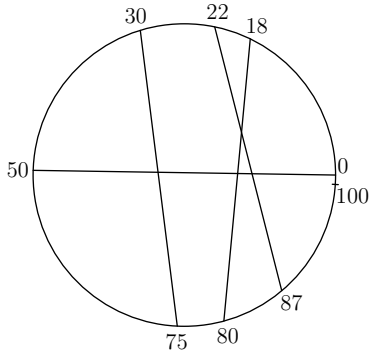
Implementácie všeobecne známych algoritmov a dátových štruktúr môžete nájsť vo vzorových riešeniach kategórií Z a O starších ročníkov KSP alebo aj na internete.

1. Trasy lodí

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

Zadanie úlohy bolo jednoduché: máme kruh a ním prechádzajúce priamky, spočítajte dvojice priamiek, ktoré sa vo vnútri kruhu križujú. Celé je to ešte zjednodušené tým, že priesečníky priamiek s kruhom sú všetky navzájom rôzne.

Za túto úlohu je hanba mať nulu, pretože riešenie hrubou silou – presnejšie, v čase $\Theta(n^2)$ – je veľmi jednoduché. Pre každú dvojicu priamiek vieme v konštantnom čase spočítať, či sa križujú alebo nie: Ak máme priamku spájajúcu body a a b, pričom $a < b$, a druhú priamku spájajúcu body c a d, tak sa križujú vo vnútri kruhu vtedy a len vtedy, ak práve jedno z čísel c a d leží v intervale (a, b). Rozmyslite si to pri pohľade na nasledujúci obrázok:



Veselšie to samozrejme bude akonáhle sa pokúsime o riešenie s lepšou časovou zložitosťou.

Základná myšlienka nášho riešenia pritom nebude vôbec zložitá. Mnohé problémy na kruhu sa dajú previesť na problémy na priamke, a tie sa väčšinou dajú riešiť jednoduchšie. Presne o to sa pokúsime aj v našom vzorovom riešení.

Predstavme si, že sme zobrali nožnice a prestrihli kruh medzi ľubovoľnými dvomi z bodov zadaných na vstupe – napríklad medzi bodmi s číslom 0 a r. (Na našom obrázku teda medzi bodmi 0 a 100.) Kruh teraz chytíme za oba konce a vystrieme ho na úsečku. Pre kruh z príkladu vyššie by sme takto dostali nasledovnú úsečku:

Každá z pôvodných priamok (presnejšie, tetív našej kružnice) sa teraz zmenila na nejaký interval na našej úsečke. Dôležité je uvedomiť si, že tieto intervaly naďalej nesú všetku informáciu potrebnú na vyriešenie úlohy. (Totiž nijak sme nezmenili čísla zo vstupu, len sa na ne inak pozeráme.) A navyše túto informáciu z nich vieme ľahko získať:

- Dvojica tetív zodpovedajúca disjunktným intervalom sa nepretína.
- Dvojica tetív zodpovedajúca intervalom z ktorých je jeden celý vnútri druhého sa nepretína.

- Všetky ostatné dvojice tetív sa pretínajú.

Inými slovami, stačí nám zistiť počet dvojíc intervalov, ktoré sa len čiastočne prekrývajú: teda idúc zľava doprava najskôr začne prvý interval, potom začne druhý, potom skončí prvý a až po ňom skončí druhý interval.

Tento počet dvojíc vieme ľahko určiť zametaním. Začneme tým, že si všetky začiatky a konce našich intervalov uložíme do jedného poľa a toto pole usporiadame. V tomto poradí ich teraz budeme spracúvať.

A čo sa bude diať počas tohto spracúvania? Budeme si (v nejakej šikovnej podobe, ktorú upresníme neskôr) pamätať, ktoré intervaly sú momentálne otvorené – teda množinu intervalov, ktorých začiatok sme už spracovali ale koniec ešte nie. Vždy, keď spracujeme ďalší začiatok intervalu, pribudne nám nejaký interval do tejto množiny, a vždy, keď spracujeme koniec, tak nám z nej jeden interval ubudne.

Navyše vždy, keď spracúvame koniec nejakého intervalu, započítame nejaké dvojice čiastočne sa prekrývajúcich intervalov – tie, v ktorých je interval, ktorý práve skončil, "prvý". Koľko je takých dvojíc? "Druhým" intervalom v každej takejto dvojici je určite niektorý z intervalov, ktoré sú práve otvorené. Treba si ale uvedomiť, že my nechceme úplne všetky takéto intervaly – len tie z nich, ktoré začali neskôr ako náš interval, ktorého koniec práve spracúvame.

Potrebujeme teda vedieť efektívne odpovedať na otázky nasledujúceho typu: "Koľko spomedzi intervalov ktoré sú práve otvorené, má začiatok medzi x a y?" Toto vieme spraviť veľa rôznymi spôsobmi. Napríklad môžeme použiť vyvažovaný binárny vyhľadávací strom, v ktorého vrcholoch si pamätáme začiatky aktuálne otvorených intervalov, a navyše informáciu o tom, koľko vrcholov stromu sa pod ním nachádza. Pomocou takejto dátovej štruktúry vieme ľubovoľnú otázku vyššie uvedeného typu zodpovedať v logaritmickom čase.

Existujú však aj stručnejšie možnosti implementácie. K tým nám môže pomôcť napríklad to, že si uvedomíme, že na konkrétnych súradniciach začiatkov a koncov našich intervalov vôbec nezáleží. Keď už ich raz usporiadame, riešenie sa vôbec nezmení, ak ich následne prečíslujeme na 1 až $2n$. No a udržiavať si podmnožinu množiny $\{1, \ldots, 2n\}$ je už výrazne ľahšie ako robiť to vo všeobecnosti. Môžeme na to použiť napríklad intervalový strom (ľudovo nazývaný intervaláč) alebo Fenwickov strom (u nás ľudovo nazývaný fínsky strom, pozri http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees). Ten druhý používame aj v našej nižšie uvedenej implementácii.

Všetky vyššie popísané riešenia majú zjavne časovú zložitosť $\Theta(n \log n)$ a pamäťovú $\Theta(n)$.

Listing programu (C++)

```
#include <algorithm>
#include <iostream>
#include <map>
#include <set>
#include <vector>
using namespace std;

struct FenwickTree {
    int fsize;
    vector<int> F;

    FenwickTree(int sz) { fsize=1; while (fsize<sz) fsize <<= 1; F.clear(); F.resize(fsize+1); }

    void update(int x, int d) { while (x <= fsize){ F[x]+=d; x+=x&-x; } }

    int sum(int x1, int x2) { // sucet v zlava-otvorenom intervale (x1,x2]
        int res = 0;
        while (x2) { res += F[x2]; x2 -= x2 & -x2; }
        while (x1) { res -= F[x1]; x1 -= x1 & -x1; }
        return res;
    }
};

struct event {
    long long kde;
    int id, typ;
    event(long long kde, int id, int typ) : kde(kde), id(id), typ(typ) {}
};

bool operator< (const event &A, const event &B) { return A.kde < B.kde; }

int main() {
    int N; cin >> N;
    long long R; cin >> R;

    vector<long long> Z(N); // Z[i] je zaciatok intervalu cislo i
    vector<event> events;
    for (int n=0; n<N; ++n) {
        long long z, k; cin >> z >> k; if (z>k) swap(z,k);
        Z[n] = z;
        events.push_back( event(z,n,+1) );
        events.push_back( event(k,n,-1) );
    }
    sort( events.begin(), events.end() );

    for (int n=0; n<2*N; ++n) { // precislujeme suradnice na 1 az 2N
        events[n].kde = n+1;
```

```
        if (events[n].typ == +1) Z[ events[n].id ] = n+1;
    }

    FenwickTree F(2*N+7);
    long long answer = 0;
    for (auto e : events) {
        if (e.typ == +1) {
            F.update(e.kde,+1);
        } else {
            int my_start = Z[e.id];
            F.update(my_start,-1);
            answer += F.sum(my_start,e.kde);
        }
    }
    cout << answer << endl;
}
```

Bonus na záver

Štandardný `set` v C++ veci potrebné na riešenie tejto úlohy robiť nevie. Konkrétne, nevie nám odpovedať na otázku, koľko spomedzi v ňom uložených prvkov leží v danom intervale. Netreba však hneď implementovať vlastný strom. V novších verziách g++ kompilátora nájdeme aj takzvané "policy-based data structures" a medzi nimi aj stromy, ktoré potrebnú fičúriu majú. A s nimi je už riešenie našej úlohy hračkou.

Listing programu (C++)

```
#include <algorithm>
#include <iostream>
#include <map>
#include <set>
#include <vector>
using namespace std;
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

typedef tree< long long, null_type, less<long long>, rb_tree_tag, tree_order_statistics_node_update > ordered_set;

struct event {
    long long kde, zac;
    int id, typ;
    event(long long kde, long long zac, int id, int typ) : kde(kde), zac(zac), id(id), typ(typ) {}
};

int main() {
    int N; cin >> N;
    long long R; cin >> R;

    vector<event> events;
    for (int n=0; n<N; ++n) {
        long long z, k; cin >> z >> k; if (z>k) swap(z,k);
        events.push_back( event(z,z,n,+1) );
        events.push_back( event(k,z,n,-1) );
    }
    sort( events.begin(), events.end(), [](const event &A, const event &B) { return A.kde < B.kde; } );

    long long answer = 0;
    ordered_set open;
    for (auto e : events) {
        if (e.typ == +1) {
            open.insert(e.zac);
        } else {
            open.erase(e.zac);
            answer += open.order_of_key(e.kde) - open.order_of_key(e.zac);
        }
    }
    cout << answer << endl;
}
```

(Tento strom podporuje aj operáciu inverznú k `order_of_key`: metóda `find_by_order(x)` vráti iterátor na x-tý najmenší prvok, číslujúc od nuly. Obe operácie bežia v čase logaritmickom od počtu uložených prvkov.)

Bonus za záverom

Ak by niekedy došlo na najhoršie a bolo naozaj treba od základov implementovať vlastný vyvažovaný strom, dôležité je dodržať dve zásady:

- nepodľahnúť panike
- vedieť, aký strom implementovať

Hrdinom dnešných dní je treap (http://en.wikipedia.org/wiki/Treap): strom, ktorý na to, aby bol *s veľkou pravdepodobnosťou* vyvážený, používa náhodné čísla. Keď porozumiete tomu, ako treap funguje, je jeho implementácia (spravená správnym spôsobom) až prekvapivo stručná a takmer bez špeciálnych prípadov. Trik na dobrú implementáciu je nasledovný:

- jediným rekurzívnym prechodom zhora dole vieme treap rozbiť na dva menšie (`split`)
- taktiež jediným prechodom zhora dole vieme tie dva menšie spojiť späť (`merge`)
- vkladanie prvku aj výber prvku vieme triviálne realizovať pomocou `split` a `merge`
- čokoľvek ďalšie dorobíme rovnako ako by sme to robili v nevyvažovanom strome

Tu je v celej jej kráse implementácia, ktorá sa dá priamo použiť v predchádzajúcom riešení namiesto `ordered_tree`.

Listing programu (C++)

```
struct treap {
    struct treap_node {
        long long x; // prvok ktory si pamatame
        int y;       // nahodna priorita
        int size;    // velkost podstromu pod tymto vrcholom
        treap_node *l, *r;

        treap_node(long long x) : x(x), y(rand()), size(1), l(NULL), r(NULL) {}
        ~treap_node() { delete l; delete r; }

        void refresh() { size = 1 + (l ? l->size : 0) + (r ? r->size : 0); }
    };

    void split(treap_node *root, long long x, treap_node* &l, treap_node* &r) {
        // rozdeli treap na dva: lavy obsahujuci hodnoty <x a pravy obsahujuci >=x
        l = r = NULL;
        if (!root) return;
        if (root->x < x) { split(root->r, x, root->r, r); l=root; }
        else             { split(root->l, x, l, root->l); r=root; }
        root->refresh();
    }

    treap_node* merge(treap_node *l, treap_node *r) {
        // undo split
        if (!l || !r) return l ? l : r;
        treap_node *p;
        if (l->y < r->y) { p=l; p->r = merge(p->r,r); } else { p=r; p->l = merge(l,p->l); }
        p->refresh();
        return p;
    }

    treap_node *root;

    treap() : root(NULL) {}
    ~treap() { delete root; }

    void insert(long long x) {
        treap_node *l, *r;
        split(root,x,l,r);
        root = merge(merge(l,new treap_node(x)),r);
    }

    void erase(long long x) {
        treap_node *l, *m, *r;
        split(root,x,l,r);
        split(r,x+1,m,r);
        if (m) delete m;
        root = merge(l,r);
    }

    int order_of_key(long long x) {
        treap_node *l, *r;
        split(root,x,l,r);
        int answer = l ? l->size : 0;
        root = merge(l,r);
        return answer;
    }
};
```

návod písal Tomi

2. Treba vysávať

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

Predstavme si, že nemáme strom, ale len čiaru. O každom políčku (každej chodbe) chceme rýchlo zisťovať, koľkokrát sme ju povysávali, a navyše chceme rýchlo inkrementovať viacero za sebou idúcich chodieb. Na to môžeme použiť intervalový strom, ktorý oboje zvláda v čase $O(\log n)$.

Čo s tým, že pracujeme na strome? Nevadí – použijeme *heavy light dekompozíciu*, ktorá náš strom rozreže na pásiky. Na každom pásiku spravíme intervaláč a sme hotoví.

Heavy light dekompozícia funguje tak, že hrany rozdelí na ťažké a ľahké. Každý vnútorný vrchol bude zo svojich synov spojený ťažkou hranou iba s jediným, a to s tým, ktorý má najviac potomkov. (Ak sú takí viacerí, nejakého si vyberieme.) So všetkými ostatnými synmi bude spojený ľahkou hranou. Takže z každého vrcholu vedú najviac dve ťažké hrany – možno jedna do niektorého syna, a možno jedna do rodiča. Tieto pásiky voláme ťažké cesty.

Takéto rozdelenie hrán má veľmi peknú vlastnosť: Keď chceme prejsť z nejakého vrcholu do koreňa, cestou uvidíme najviac $\log_2 n$ ľahkých hrán. To preto, že keď prejdeme po ľahkej hrane zo syna s do rodiča r, ten rodič r musí mať nejakého iného syna t, s ktorým je spojený ťažkou hranou. Takže t má aspoň toľko potomkov, ako s, a r má aspoň dvakrát toľko potomkov.

Preto platí, že každá cesta z a do b sa prekrýva s najviac $O(\log n)$ ťažkými cestami. Každú ťažkú cestu vybavíme v $O(\log n)$ (na každej ťažkej ceste máme spravený intervaláč) a každú ľahkú cestu vybavíme priamo, veď ich je málo. Takže inkrementovať každú hranu od a po b dokážeme v čase $O(\log^2 n)$, a hodnotu hrany zistíme v $O(\log n)$.

návod písal Jano

3. Tesná medzera v stene

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

V prvom rade treba vymyslieť nutnú a postačujúcu podmienku, kedy sa had dokáže prepchať cez škáru.

Had sa dokáže prepchať cez škáru práve vtedy, keď pre každý bod hada existuje priamka, ktorá prechádza týmto bodom a hada nikde inde nepretína. Zamyslite sa prečo. Všimnite si tiež, že vďaka nekolinearite každej trojice bodov nemusíme rozlišovať prípady "pretína" a "dotýka".

Had sa dokáže prepchať cez škáru práve vtedy, keď sa pri ľubovoľnom rozseknutí hada v nejakom jeho bode na dve časti konvexné obaly týchto častí neprekrývajú (ich prienik má nulový obsah, resp. majú spoločný len jeden bod alebo úsečku). Zamyslite sa, ako toto vyplýva z predošlého pozorovania. Túto podmienku stále nevieme algoritmicky overovať, pretože možných rozseknutí je nekonečne veľa. Preto treba pokračovať v úvahách.

Had sa dokáže prepchať cez škáru práve vtedy, keď sa pri ľubovoľnom rozseknutí hada v nejakom jeho vrchole (to sú body, kde sa láme jeho telo) konvexné obaly častí neprekrývajú. Môžeme si uvedomiť, že aj prvú podmienku (s priamkami) stačí overovať len pre vrcholy hada.

Ponúka sa teda myšlienkovo jednoduché, ale implementačne náročné riešenie úlohy. Postupne vyskúšame rozseknúť hada v každom z n vrcholov, pri každom rozseknutí spočítame konvexný obal predku (všetko po bod rozseknutia vrátane) a zadku (všetko od bodu rozseknutia vrátane) hada a spočítame, či sa konvexné obaly prekrývajú. Riešenie by malo zložitosť $O(n^2 \log(n))$.

Úloha sa dá naprogramovať aj oveľa príjemnejšie, pretože v skutočnosti nemusíme konštruovať konvexné obaly, ale stačí pre každý bod rozseknutia A spočítať interval uhlov, pod ktorými vidíme predok hada z A a zadok hada z A. Ak sa intervaly prekrývajú alebo ak je niektorý interval väčší ako 180 stupňov, had sa neprepchá cez stenu.

Samozrejme, nebudeme počítať uhly, ale iba nájdeme najľavejší a najpravejší bod predku/zadku hada pomocou vektorového súčinu, vďaka čomu všetky výpočty ostanú v celých číslach. Tak sa dá úloha vyriešiť elegantne na pár riadkov, v čase $O(n^2)$. Odporúčame premyslieť si implementáciu pred tým, ako začnete programovať.

návod písal Jano

4. Tvorba prútikov

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

Namiesto toho, aby sme hľadali T v $p^k(S)$, budeme hľadať $q^k(T)$ v S, pričom q bude také zobrazenie, aby sa $q^k(T)$ nachádzalo v S práve vtedy, keď T v $p^k(S)$. Výhodou tohto prístupu je, že q bude reťazec skracovať približne na polovicu, čím dosiahneme dobrú časovú zložitosť, zatiaľ čo p reťazec predlžoval, čo by viedlo k pomalým časovým zložitostiam.

Ukážeme si to na príklade, predstavme si, že vstup je: T = „baaaaaaabaa“ a S = „ababa“. Správna odpoveď je 2, lebo p^2(„ababa“) = „aaaaaaa*baaaaaaabaa*aa”. Odpoveď však vieme spočítať aj tzv. redukovaním, ktoré bližšie popíšeme nižšie:

"baaaaaaabaa" sa nenachádza v "ababa", preto redukujeme T.

"baaaba" sa nenachádza v "ababa", preto redukujeme "baaaba".

"bab?" sa nachádza v "ababa", preto je odpoveď 2. "?" je žolík, môže pasovať aj na "a" aj na "b".

Redukcia nejakého slova A je najkratšie také slovo B, že $p(B)$ obsahuje A. Napríklad p(„baaaba“) = „abaaaaaaabaa“, čo obsahuje "baaaaaaabaa", preto "baaaba" je redukciou "baaaaaaabaa". V tomto prípade je to jediná možná redukcia, ale vo všeobecnosti nemusí byť redukcia jednoznačne určená. Napríklad "baba" a "babb" sú dve možné redukcie "baaaba". Na druhej strane, nie každé slovo musí mať redukciu. Napríklad neexistuje redukcia slov "baab" či "bb".

Dôležité v tejto úlohe je zamyslieť sa, ako redukcia funguje a skúsiť si ju spraviť ručne na papieri pre nejaké slová. Tak si všimneme, že redukcia sa správa pomerne pekne. Jednoduchým lineárnym prechodom vieme spočítať redukciu nejakého slova, alebo povedať, že žiadna neexistuje. Ak nie je redukcia jednoznačná (to

je vtedy, keď za posledným béčkom je nepárny počet áčok), tak vždy je nejednoznačný práve jeden posledný znak, ktorý môžeme označiť ako otáznik, ako sme to videli v príklade. Redukcie slov s otáznikom na konci budu mať zasa (jeden) otáznik na konci. Trocha si treba dať pozor na redukcie samých áčok. Redukcia "aaaaa" je "aa?", prípadne redukcia "a" je "?".

Naprogramujeme teda redukujúcu funkciu a úlohu vyriešime tak, že dokola redukujeme vstupné T, až kým sa nebude redukované T vyskytovať v S (vtedy vypíšeme potrebný počet redukcií) alebo sa nám redukcia nepodarí (vtedy vypíšeme -1).

Ako zistíme, či sa daný reťazec T nachádza v S? Jednoduchým KMP algoritmom. V prípade, že hľadaný text končí znakom "?", tak hľadáme T bez posledného znaku v S bez posledného znaku. Časová zložitosť celého algoritmu je $O(n \log n)$ kde n je dĺžka vstupu. Spravíme totiž najviac $O(\log n)$ redukcií.

návod písal mišof

5. To kto ešte stále píše perom?

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

Čo už s takouto úlohou?

Jedna rozumná možnosť je nájsť spôsob ako z nej bezbolestne požmýkať čo najviac bodov. Celkom dosť sa toho dá dosiahnuť jednoduchými algoritmickými trikmi. Napríklad vieme spočítať počty uzavretých oblastí (8 má dve, 6 má jednu v dolnej časti, 4 a 9 majú jednu v hornej časti, 0 má jednu na celú výšku) a tiež počty miest kde sa čiara vetví/pretína. Takéto niečo nebude síce 100-percentne úspešné, ale bodov to dá dosť a kódi sa to ľahko.

Lepšiu úspešnosť by sme vedeli dostať výpočtom *korelácie*: zoberieme obrázok, ktorý sme dostali na vstupe, porovnáme ho s 10 000 známymi bitmapami, a odpovieme podľa tej, na ktorú sa najviac podobá. Takýto prístup však má dva problémy: porovnávanie s 10 000 bitmapami bude asi trošku pomalé, no a hlavne by sme potrebovali všetky napchať do zdrojového kódu nášho programu.

Oba tieto problémy vieme vyriešiť tak, že najskôr si doma *natrénujeme* náš program na obrázkoch, ktoré máme – inými slovami, nájdeme vhodnú funkciu, do ktorej keď vopcháme bitmapu tak vypadne číslo na nej. Odovzdaný program bude potom namiesto tých 13 MB dát, ktoré sme dostali, obsahovať len popis tejto funkcie.

Akú formu môže mať táto funkcia? Dalo by sa napríklad hľadať rôzne váhy, ktorými budeme násobiť farbu rôznych políčok. Existujú však aj omnoho lepšie možnosti. Jednou z nich je perceptrón – asi najjednoduchšia forma umelej neurónovej siete. Pri trénovaní perceptrónu v podstate hľadáme naraz 10 funkcií. Každá z nich sa pozerá na obrázok a snaží sa rozhodnúť, či to vyzerá alebo nevyzerá byť jedna z našich 10 cifier.

V skutočnosti sa dokonca presne tie dáta, ktoré boli použité v našej úlohe, používajú ako testovacie dáta pre porovnávanie úspešnosti rôznych prístupov strojového učenia. Najlepšie prístupy majú pri rozpoznávaní rovnakú úspešnosť ako ľudia – teda v princípe všetko rozpoznajú správne, až na pár vstupov u ktorých naozaj nie je jasné, čo tým chcel básnik povedať.

Viac sa o tom všetkom dočítate napríklad na tejto stránke.