P-II-1 Vzestup a pád

První řešení

Nechť b je rok, v němž Kocourkov dosáhl úplně největší populace. Jestliže v nějakém roce a předcházejícím roku b dosáhl vyšší populace než v nějakém roce c následujícím po roce b, pak trojice a, b a c splňuje požadavky zadání; abychom rozhodli, zda takové roky a a c existují, stačí nám znát maximální populaci v letech 1 až $b-1$ a minimální populaci v letech $b+1$ až N.

Jestliže takto rovnou nenalezneme řešení, pak byla populace před rokem b vždy nižší než populace po roce b, hledaná trojice roků tedy musela buď celá nastat před rokem b, nebo celá po roce b. To nám dává přirozený rekurzivní algoritmus:

```
/* Nalezne trojici a<b<c s vlastnostmi ze zadání tž. od <= a, b, c <= do.
   Vrací true existuje-li taková trojice, false jinak.  */
bool vzestup_a_pad (int od, ind do)
{
  if (do <= od + 1)
    return false;

  b = pozice maxima populace z intervalu od, do;
  a = pozice maxima populace z intervalu od, b - 1;
  c = pozice minima populace z intervalu b + 1, do;
  if (populace[c] < populace[a])
    {
      vypiš a, b, c;
      return true;
    }

  return vzestup_a_pad (od, b - 1) || vzestup_a_pad (b + 1, do);
}
```

Počet rekurzivních volání funkce `vzestup_a_pad` je nejvýše N – stačí nahlédnout, že každé číslo ze vstupu je maximem v nejvýše jednom z intervalů, na které je tato funkce volaná. Potřebujeme ale umět efektivně hledat pozice maxim a minim v intervalech.

Jednoduché řešení je si předpočítat pozici minima a maxima pro každý interval, jehož délka je mocnina 2. Takových intervalů je $\mathcal{O}(N \log N)$ a jejich minima a maxima snadno spočítáme se stejnou časovou složitostí (minimum z intervalu od a do $a+2^k-1$ je rovno minimu z intervalů od a do $a+2^{k-1}-1$ a od $a+2^{k-1}$ do $a+2^k-1 = (a+2^{k-1})+2^{k-1}-1$, máme-li tedy minima již předpočítaná pro intervaly délky 2^{k-1}, můžeme ho určit v konstantním čase). Chceme-li nyní zjistit minimum z intervalu od a do b, stačí najít největší mocninu dvojky x menší nebo rovnu $b-a+1$ (kterou můžeme mít také předpočítanou) a vrátit menší z minim intervalů od a do $a+x-1$ a od $b-x+1$ do b. Pozici minima a analogicky maxima z intervalu tedy dokážeme určit v konstantním čase.

Celková časová a paměťová složitost tohoto řešení tedy bude $\mathcal{O}(N \log N)$, dominuje ji složitost předzpracování pro vytvoření datové struktury popsané v předchozím odstavci. Poznamenejme, že existuje obdobná datová struktura, kterou lze vytvořit v lineárním čase, čímž lze časovou složitost vylepšit na $\mathcal{O}(N)$; nicméně tato datová struktura je podstatně složitější.

Druhé řešení

Existuje ale i jednodušší řešení. Budeme postupně procházet posloupnost p_1, …, p_N populací v jednotlivých rocích. Nechť nyní zpracováváme rok x a zajímá nás, zda tento rok může hrát roli roku c v nějakém řešení. To se stane právě tehdy, když existují nějaké roky $a < b$ před x takové, že $p_a < p_b$ a zároveň $p_x < p_a$. Pro ověření této podmínky zjevně stačí znát roky a a b takové, že $a < b < x$ a $p_a < p_b$ a populace p_a je největší možná mezi dvojicemi a a b splňujícími tyto podmínky.

Budeme si tedy průběžně udržovat informaci o takových rocích a a b a jejich populacích p_a a p_b. Jak se tato informace může změnit při zpracování roku x? Je možné, že existuje nějaký rok y přecházející roku x takový, že $p_y > p_a$, ale mezi x a y není nikdy populace vyšší než p_y, takže y zatím nesplňuje podmínky pro možnou volbu roku a. Jestliže ale $p_x > p_y$, pak bychom si chtěli místo roků a a b od nynějška pamatovat roky y a x.

Abychom toto mohli dělat efektivně, budeme si kromě roku b také pamatovat ostatní zatím zpracované roky y, v nichž byla populace vyšší než v roce a. Povšimněme si, že takové roky se nutně v posloupnosti vyskytují v klesajícím pořadí populací, jinak by některý z nich musel hrát roli a. Bez újmy na obecnosti proto můžeme předpokládat, že b je poslední z takových roků (a má tedy mezi nimi nejmenší populaci). Místo jednoho roku b si proto budeme pamatovat zásobník B roků, v nichž je populace vyšší než v roce a, setřízený tak, že populace roku na vrcholu zásobníku B je nejmenší.

Když zpracováváme rok x, nejprve zkontrolujeme, zda $p_x < p_a$; je-li tomu tak, pak a, rok na vrcholu zásobníku B a rok x tvoří hledané řešení. Jestliže $p_x > p_a$, pak zkontrolujeme, zda nějaké roky na zásobníku B mají populaci menší než p_x. Jestliže ano, pak si ten z nich s největší populací zapamatujeme jako rok a a roky s menší populací vyhodíme z B; povšimněme si, že vzhledem k uspořádání roků v zásobníku projdeme v B nejvýše o dva prvky víc, než kolik jich vyhodíme, časová složitost tohoto kroku je tedy úměrná počtu smazaných prvků. Každopádně pak vložíme x na zásobník B, jelikož jeho populace je vyšší než populace v roce p_a (ať už jsme a měnili nebo ne).

Jelikož popsaný algoritmus pouze jednou projde vstup a každé číslo nejvýše jednou přidá a odebere ze zásobníku B, časová i paměťová složitost je $\mathcal{O}(N)$.

```
#include <stdio.h>

#define MAXN 10000000
static int Bpopulace[MAXN];
static int Bpozice[MAXN];
static int velikostB;
static int Apopulace, Apozice;
```

```
int main (void)
{
  int N, i, x;

  scanf ("%d", &N);
  Apopulace = -1;
  Apozice = -1;
  velikostB = 0;
  for (i = 0; i < N; i++)
    {
      scanf ("%d", &x);
      if (x < Apopulace)
        {
          printf ("%d %d %d\n", Apozice, Bpozice[velikostB - 1], i + 1);
          return 0;
        }
      while (velikostB && Bpopulace[velikostB - 1] < x)
        {
          velikostB--;
          Apopulace = Bpopulace[velikostB];
          Apozice = Bpozice[velikostB];
        }
      Bpopulace[velikostB] = x;
      Bpozice[velikostB] = i + 1;
      velikostB++;
    }

  printf ("nelze\n");

  return 0;
}
```

P-II-2 Převratná novinka

Z učebního textu o stromochodech pravděpodobně znáte základní grafovou terminologii (vrcholy, hrany, cesty), kterou budeme používat i v tomto řešení (křižovatky jsou vrcholy, ulice jsou hrany).

Jak získat pět bodů?

Ukážeme si, jak vyřešit úlohu v čase $\mathcal{O}(m \log n)$ za předpokladu, že je optimální pouze jednou změnit dopravní prostředek. Budeme předpokládat, že Norbert vyjde z domova pěšky a někde cestou přesedne na koně (opačná situace je symetrická a náš algoritmus může vyzkoušet obě a vrátit lepší z nich).

Jako přímočaré řešení by se nabízelo zjistit si délku nejkratší pěší cesty z Norbertova domova do každého vrcholu s koňostavem a potom délku nejkratší cesty na koni z každého vrcholu s koňostavem na radnici, a potom vybrat takový koňostav, že součet délek jeho příslušných dvou cest je co nejmenší. Na hledání nejkratších cest použijeme *Dijkstrův algoritmus,** který nám dokáže zjistit délky nejkratších cest z jednoho vrcholu do všech ostatních v čase $\mathcal{O}(m \log n)$.

Takové řešení by však rozhodně neseběhlo dostatečně rychle, jelikož musíme spustit Dijkstrův algoritmus z každého vrcholu z koňostavem, kterých může být až n.

* Viz kapitola v Kuchařce KSP: *https://ksp.mff.cuni.cz/kucharky/halda-a-cesty/*

Všimněme si ale, že pro každý koňostav nás zajímá pouze nejkratší cesta na radnici. A tedy místo toho, abychom spouštěli Dijsktrův algoritmus z každého vrcholu s koňostavem, ho můžeme spustit z vrcholu s radnicí a zaznamenat si všechny nejkratší cesty do vrcholů s koňostavy. Tím již dostaneme časovou složitost $\mathcal{O}(m \log n)$ a kýžených pět bodů.

A co sedm bodů?

Nyní už je možné, že v optimálním řešení bude potřeba několikrát změnit dopravní prostředek. Máme ovšem slíbeno, že míst, kde bude možné dopravní prostředek měnit, bude málo.

Představme si nyní, že máme optimální řešení $\mathcal{O}$. Nejprve si uvědomme, že se v $\mathcal{O}$ neopakuje žádný vrchol s koňostavem – délky hran jsou kladná celá čísla, a kdyby se tedy v $\mathcal{O}$ dvakrát vyskytl vrchol x_i, v němž je koňostav, mohli bychom odebrat celou část cesty mezi těmito dvěma výskyty x_i, čímž bychom cestu zkrátili, což je spor s tím, že $\mathcal{O}$ je optimální.

Označme $\tilde{O}$ posloupnost vrcholů s koňostavy v pořadí, v jakém se vyskytují v $\mathcal{O}$ (tedy speciálně $\tilde{O}$ začíná vrcholem, kde Norbert bydlí, a končí vrcholem, kde se nachází radnice). Z předchozího odstavce víme, že se v $\tilde{O}$ neopakují vrcholy. Navíc v každém vrcholu $\tilde{O}$ můžeme libovolně změnit dopravní prostředek. To znamená, že jsou-li vrcholy x_1, x_2 sousední v $\tilde{O}$, tak mezi nimi jdeme nejkratší možnou cestou při použití pouze jednoho dopravního prostředku.

To vede k řešení se složitostí $\mathcal{O}(sm \log n + s^2 \log s)$. Označme si jako G graf, který jsme dostali na vstupu. Nejprve si z každého vrcholu x s koňostavem spustíme dvakrát Dijkstrův algoritmus – jednou chodíme pouze pěšky a jednou jezdíme pouze na koni – a pro každý vrchol y s koňostavem si zaznamenáme délku nejkratší cesty mezi x a y (tj. tu kratší z cest pěšky resp. na koni). Nyní si můžeme vytvořit nový pomocný graf $\tilde{G}$, jehož vrcholy budou pouze vrcholy s koňostavy, a mezi každé dva vrcholy natáhneme hranu délky odpovídající délce nejkratší cesty v kocourkovském grafu G (nebo žádnou, pokud mezi těmito dvěma vrcholy nevede cesta). To stihneme v čase $\mathcal{O}(sm \log n)$ – $2s$-krát spustíme Dijkstrův algoritmus, dvakrát pro každý vrchol s koňostavem, konstrukce grafu $\tilde{G}$ je pak již lineární.

A nyní již jen stačí najít nejkratší cestu mezi Norbertovým domovem a radnicí v grafu $\tilde{G}$ (což nám opět pomocí Dijkstrova algoritmu zabere $\mathcal{O}(s^2 \log s)$ času, jelikož hran v $\tilde{G}$ může být až s^2). Tato nalezená cesta jistě odpovídá nějaké cestě s přesedáním v kocourkovském grafu. Naopak, jak jsme si již rozmysleli, optimální řešení $\mathcal{O}$ odpovídá nějaké posloupnosti vrcholů s koňostavy, kde se mezi sousedními vrcholy přesunujeme bez změny dopravního prostředku, tedy přesně cestě v grafu $\tilde{G}$.

Optimální řešení

Rádi bychom pustili Dijkstrův algoritmus jen jednou. Ale to jistě nepůjde přímo na graf G, jelikož algoritmus neví, kterou délku hran má používat. Potřeboval by v každém kroku vědět, zda z aktuálního vrcholu má odejít po hranách pěšky, anebo odjet na koni.

A to přesně uděláme. Vytvoříme graf G' tak, že za každý vrchol $v \in G$ přidáme do G' jeho dvě kopie, v_p a v_k, kde v_p znamená, že jsme ve vrcholu v pěšky a v_k znamená, že tam jsme na koni. Pokud je v G hrana uv, tak do G' umístíme hrany u_pv_p a u_kv_k, přičemž délka u_pv_p bude rovna délce uv, když se po ní jde pěšky, a délka u_kv_k bude rovna délce uv na koni.

Navíc ještě musíme nějak v G' zachytit přesedání a to uděláme tak, že pro každý vrchol $v \in G$, v němž je koňostav, přidáme do G' hranu v_pv_k a dáme jí délku nula.

Každá cesta v G' jednoznačně odpovídá cestě v G s přesedáním a naopak. Ale na G' nyní už můžeme spustit Dijkstrův algoritmus (z libovolné kopie Norbertova domu) a vzdálenost libovolné kopie radnice, kterou nám algoritmus spočítá, bude naše hledaná odpověď.

```
#include <cstdio>
#include <queue>
#include <vector>
using namespace std;

#define maxN 100005

int n, m, d, r;
vector<int> hrany[2 * maxN];
vector<int> delka[2 * maxN];
long long vzdalenost[2 * maxN];
bool zpracovany[2 * maxN];

int main() {
  scanf("%d%d%d%d", &n, &m, &d, &r);
  --d; --r;  // Vrcholy interně číslujeme od nuly.
  for (int i = 0; i < m; ++i) {
    int a, b, pesky, konem;
    scanf("%d%d%d%d", &a, &b, &pesky, &konem);
    --a; --b;  // Vrcholy interně číslujeme od nuly.
    // Vrchol 2*a odpovídá vrcholu pro chození pěšky, 2*a+1 pro jízdu na koni.
    for (int k = 0; k < 2; ++k) {
      int delka_hrany = (k == 0) ? pesky : konem;
      hrany[2 * a + k].push_back(2 * b + k);
      delka[2 * a + k].push_back(delka_hrany);
      hrany[2 * b + k].push_back(2 * a + k);
      delka[2 * b + k].push_back(delka_hrany);
    }
  }
  int s;
  scanf("%d", &s);
  for (int i = 0; i < s; ++i) {
    int x;
    scanf("%d", &x);
    --x;
    hrany[2 * x].push_back(2 * x + 1);
    delka[2 * x].push_back(0);
    hrany[2 * x + 1].push_back(2 * x);
    delka[2 * x + 1].push_back(0);
  }
```

```
  // Dijkstrův algoritmus na hledání nejkratší cesty.
  for (int i = 0; i < 2 * n; ++i) {
    zpracovany[i] = false;
    vzdalenost[i] = -1;  // K vrcholu se zatim neda dostat.
  }

  // Do haldy dáváme dvojice (-vzdálenost, index vrcholu), protože
  // priority_queue třídí primárně podle první slozky a preferuje větší čísla.
  priority_queue<pair<long long, int> > fronta;
  fronta.push(make_pair(0, 2 * d));
  vzdalenost[2 * d] = 0;
  while (!fronta.empty()) {
    int u = fronta.top().second; fronta.pop();
    if (u == 2 * r) break;  // Vrchol, do kterého se chceme dostat, už je hotový.
    if (zpracovany[u]) continue;  // Vrchol už jsme jednou zpracovali.
    zpracovany[u] = true;
    for (int i = 0; i < hrany[u].size(); ++i) {
      int v = hrany[u][i];
      // Našli jsme první nebo kratší cestu do vrcholu v.
      if (vzdalenost[v] == -1 || vzdalenost[u] + delka[u][i] < vzdalenost[v]) {
        vzdalenost[v] = vzdalenost[u] + delka[u][i];
        fronta.push(make_pair(-vzdalenost[v], v));
      }
    }
  }

  printf("%lld\n", vzdalenost[2 * r]);
  return 0;
}
```

P-II-3 Věže

Zjevně stačí umět určit nejvzdálenější viditelnou věž ve směru k radnici; pro nalezení nejvzdálenější věže v obou směrech níže popsaný algoritmus aplikujeme dvakrát, jednou ve směru k radnici a jednou ve směru od radnice, a z nalezených vzdáleností vezmeme maxima.

Nechť zadané věže mají vzdálenosti od radnice $x_1, \ldots, x_N$ a výšky $v_1, \ldots, v_N$. Nechť P_a je konvexní obal bodů $(x_1, v_1), \ldots, (x_a, v_a)$ pro nějaké a, a (x_b, y_b) je soused (x_a, v_a) na horní části hranice P_a. Povšimněme si, že vrcholy všech věží s číslem menším nebo rovným a jsou pod přímkou spojující (x_a, v_a) a (x_b, y_b), neblokují tedy výhled z věže a na věž b. Ze stejného argumentu také plyne, že věže s číslem menším než b nejsou vidět z věže a, jelikož je blokuje věž b. Proto b je nejvzdálenější věž viditelná z věže a směrem k radnici.

Stačí tedy spočítat konvexní obaly $P_1, \ldots, P_N$. Bylo by ovšem zbytečně pomalé počítat každý z těchto obalů zvlášť. Podíváme-li se na známý algoritmus pro určení konvexního obalu,* povšimneme si, že ten přidává do konstruovaného konvexního obalu body postupně dle jejich x-ové souřadnice a počítá průběžně konvexní obaly již přidaných bodů, tedy přesně to, co my potřebujeme. Stačí si tedy po každém přidaném bodu zapamatovat, do kterého bodu z něj vede hrana v horní části konvexního obalu (dolní část konvexního obalu samozřejmě vůbec nemusíme počítat).

* Viz například *https://ksp.mff.cuni.cz/kucharky/geometrie/*.

Jelikož algoritmus na nalezení konvexního obalu má (při bodech zadaných v rostoucím pořadí dle x-ové souřadnice) lineární časovou složitost, časová i paměťová složitost výsledného algoritmu je $\mathcal{O}(N)$.

```
#include <cstdio>
#include <cstdio>
#include <vector>
using namespace std;

struct bod
{
  int x, v;
  bod (int _x, int _v) : x(_x), v(_v) { }
};

static vector<bod> veze;
static vector<bod> obal;

static void
inicializuj_obal (bod &a)
{
  obal.clear();
  obal.push_back(a);
}

/* Vrátí true, jestliže bod B je nalevo od polopřímky z a_1 do a_2.  */
static bool
nalevo (bod &a1, bod &a2, bod &b)
{
  bod dp(a2.x - a1.x, a2.v - a1.v);
  bod db(b.x - a1.x, b.v - a1.v);
  return dp.x * db.v > dp.v * db.x;
}

static void
pridej_bod_do_obalu(bod &a)
{
  while (obal.size() > 1)
    {
      vector<bod>::reverse_iterator i = obal.rbegin();
      bod &posledni = *i;
      ++i;
      bod &predposledni = *i;

      if (nalevo(a, posledni, predposledni))
        break;

      obal.pop_back();
    }

  obal.push_back(a);
}

int main(void)
{
  int n;

  scanf("%d", &n);

  int vzdalenost[n];
```

```
  for (int i = 0; i < n; i++)
    {
      int x, v;
      scanf("%d%d", &x, &v);
      veze.push_back(bod(x, v));
      vzdalenost[i] = 0;
    }

  inicializuj_obal(veze[0]);
  int k = 1;
  for (vector<bod>::iterator i = veze.begin() + 1; i != veze.end(); ++i, k++)
    {
      pridej_bod_do_obalu(*i);
      bod &predposledni = *(obal.rbegin() + 1);
      int dist = i->x - predposledni.x;
      vzdalenost[k] = dist;
    }

  bod a = veze.back();
  a.x = -a.x;
  inicializuj_obal(a);
  k = n - 2;
  for (vector<bod>::reverse_iterator i = veze.rbegin() + 1;
       i != veze.rend(); ++i, k--)
    {
      a = *i;
      a.x = -a.x;
      pridej_bod_do_obalu(a);
      bod &predposledni = *(obal.rbegin() + 1);
      int dist = a.x - predposledni.x;
      if (vzdalenost[k] < dist)
        vzdalenost[k] = dist;
    }

  for (int i = 0; i < n; i++)
    printf("%d ", vzdalenost[i]);
  printf("\n");

  return 0;
}
```

P-II-4 Stromochod

a) K ověření, zda je určená množina vrcholů nezávislá, postačí obyčejné prohledání stromu do hloubky. Pokud je vrchol stromu označen, při vstupu do každého jeho syna zkontrolujeme, zda syn není také označen. To jistě zvládneme v lineárním čase s velikostí stromu.

```
type vrchol = record
   stav: 0..3;               { Kolikrát jsme ve vrcholu byli }
   vybraný: boolean;         { Patří do zadané množiny }
   ok: boolean;              { V kořeni výstup, jinde nevyužito }
end;

var otec_vybraný: boolean;
    zatím_ok: boolean;
```

```
begin
   zatím_ok := true;
   repeat
      V.stav := V.stav + 1;
      case V.stav of
         1: if ex_l then begin
               otec_vybraný := V.vybraný;
               jdi_l;
               if V.vybraný and otec_vybraný then zatím_ok := false;
            end;
         2: if ex_p then begin
               otec_vybraný := V.vybraný;
               jdi_p;
               if V.vybraný and otec_vybraný then zatím_ok := false;
            end;
         3: if ex_o then jdi_o
            else begin
               V.ok := zatím_ok;
               halt;
            end;
      end;
   until false;
end.
```

b) Nejprve „vytáhneme králíka z klobouku“ a předvedeme jednoduchý algoritmus, který najde nezávislou množinu. Později dokážeme, že tato množina je největší možná.

Strom budeme (jak jinak) prohledávat do hloubky. Kdykoliv budeme opuštět nějaký vrchol, rozhodneme se, zda ho označíme, a tím vložíme do nezávislé množiny. Opouštíme-li list, označíme ho vždy. Opouštíme-li vnitřní vrchol, označíme ho právě tehdy, není-li označený žádný ze synů. Práci si trochu zjednodušíme tím, že při prvním vstupu do vrcholu tento hned označíme a pokud se do něj vrátíme z označeného syna, značku smažeme. Program vypadá následovně.

```
type vrchol = record
   stav: 0..3;                { Kolikrát jsme ve vrcholu byli }
   vybraný: boolean;          { Patří do vybrané množiny }
end;

var syn_vybraný: boolean;

begin
   repeat
      V.stav := V.stav + 1;
      case V.stav of
         1: V.vybraný := true;
            if ex_l then jdi_l;
         2: if ex_p then jdi_p;
         3: if ex_o then begin
               syn_vybraný := V.vybraný;
               jdi_o;
               if syn_vybraný then V.vybraný := false;
            end
            else halt;
```

```
      end;
   until false;
end.
```

Algoritmus evidentně doběhne v lineárním čase a vydá nějakou nezávislou množinu. Je to ovšem typický zástupce takzvaných hladových algoritmů – vybírá si vždy lokálně nejvýhodnější volbu a neohlíží se na budoucnost. Takové algoritmy obvykle nenajdou globálně nejlepší řešení. Dokážeme, že zrovna ten náš ho najde.

Nejprve nahlédneme, že bez újmy na obecnosti můžeme do naší nezávislé množiny umístit všechny listy. Platí totiž, že libovolnou největší nezávislou množinu M můžeme předělat na jinou, stejně velkou M', která obsahuje všechny listy. Vskutku: Kdykoli v M chybí nějaký list ℓ, najdeme jeho otce p a podíváme se, zda leží v M. Kdyby neležel, mohli bychom do M přidat ℓ – tím by také vznikla nezávislá množina, takže M nebyla největší. Ve skutečné největší M tedy musí otec p ležet. Tak ho odtud odebereme a místo něj přidáme ℓ. Jak nezávislost, tak velikost množiny jsme zachovali.

Naopak vrcholy, které leží těsně nad listy, do hledané množíny jistě nepatří (jinak by nebyla nezávislá). Pokud nyní jak listy, tak tyto vrcholy odebereme, můžeme ve zbylém grafu (ten se mohl rozpadnout na více stromů) opět hledat největší nezávislou množinu. Nikdy se nám totiž nestane, že bychom vybrali nějakého souseda vrcholu vybraného v prvním kole: všichni takoví sousedé už jsou z grafu odebraní.

Nyní stačí uvědomit, že naše prohledávání do hloubky je s tímto postupem ekvivalentní. Každý list vybere, otce listu jistě nevybere (což je totéž, jako by byl odebraný), vrcholy, pod nimiž jsou jen otcové listů, opět vybere (ty se po prvním odebírání stanou listy) a tak dále.