P-II-1 Aquapark

Kdybychom chtěli řešit úlohu experimentálně a ne programem, mohli bychom postupovat následovně:

Seženeme si dostatečné množství dobrovolníků. Na začátku budou všichni jen tak sedět u vstupu na tobogany a budou pasivní. Postupně budeme zpracovávat jednotlivé zaznamenané události, kdy došlo k nasednutí člověka na tobogan. Tyto události zpracováváme v pořadí, v jakém k nim došlo. Máme-li nechat někoho nasednout na tobogan, podíváme se, zda máme k jízdě připraveného nějakého aktivního dobrovolníka. Pokud ano, jednoho z nich (je jedno, kterého) pošleme na jízdu dolů toboganem. Pokud ne, vezmeme jednoho pasivního dobrovolníka, prohlásíme ho za aktivního a pošleme dolů toboganem jeho. Dobrovolník má samozřejmě za úkol vyběhnout zpět na místo startu, jakmile jízdu na toboganu ukončí. U startu pak bude čekat připraven na další jízdu (zůstává již stále aktivní). Tvrdíme, že počet aktivních dobrovolníků na konci dne je roven minimálnímu počtu návštěvníků aquaparku, kteří mohli v ten den tobogany používat.

Popsané řešení vypadá docela dobře – nového aktivního dobrovolníka přidáme, jen když opravdu musíme. To ale ještě není důkazem správnosti. Musíme se zamyslet, zda by nebylo lepší někdy dříve přidat třeba více aktivních dobrovolníků a díky tomu někdy později ušetřit. Správnost řešení proto raději dokážeme pořádně.

Předpokládejme, že máme libovolné optimální řešení O a že máme řešení N nalezené naším algoritmem. Ukážeme, že O se dá konečným počtem kroků transformovat na N, přičemž nezměníme počet lidí, které potřebujeme.

Všimněte si první události, kdy se O a N odliší, tzn. poprvé se stane, že na nějaký tobogan nasedne v každém z těchto řešení jiná osoba. Jak se to mohlo stát? Rozebereme si několik případů:

- V řešení N jsme vytvořili nového aktivního dobrovolníka jedině tehdy, když právě nebyl žádný aktivní dobrovolník k dispozici. Jelikož řešení O se dosud s N shodovalo, v řešení O také nemáme nikoho aktivního k dispozici, takže musíme provést totéž co v N. Tento případ tedy nenastal.
- Víme tedy, že v řešení N jsme poslali na tobogan některého aktivního dobrovolníka x, který právě čekal na startu. Pokud jsme v O poslali jiného aktivního dobrovolníka y, můžeme řešení O transformovat na O' tak, že od tohoto okamžiku všechny jízdy dobrovolníka x vykoná y a naopak.
- Zbývá poslední případ: V řešení N jsme poslali na tobogan některého aktivního dobrovolníka x, zatímco v našem optimálním řešení O jsme pro tuto jízdu vytvořili nového aktivního dobrovolníka z a poslali jsme jeho. (Jinými slovy, v O jel člověk, který dosud na toboganech ještě nejel.) V tomto případě opět můžeme upravit řešení O na O' jednoduše tak, že vyměníme z a x.

Ukázali jsme, že pokud se řešení O a N liší, pak bez ohledu na to, jaký případ nastal, vždy můžeme řešení O upravit tak, abychom získali stejně dobré řešení, které se odlišuje od N až v nějakém pozdějším kroku. Po konečně mnoha opakováních výše uvedeného postupu tedy z řešení O vytvoříme naše řešení N. Tím jsme dokázali, že i naše řešení N je nutně optimální.

Zbývá navrhnout implementaci uvedeného postupu. Program jsme vytvořili tak, aby se s jeho pomocí dal navíc přímo sestrojit jeden optimální rozvrh jízd. Vstup si budeme reprezentovat pomocí tří front (pro každý tobogan jedna). Rovněž výstupy z toboganů (tj. časy, kdy z nich vystupují dobrovolníci po skončení jízdy) si udržujeme v dalších třech frontách. Nejbližší událost vždy zjistíme jako minimum z hodnot ve vstupních frontách. Při zpracování každé události se podíváme na aktuální situaci na výstupu a určíme tam nejdříve vystupujícího dobrovolníka. Jestliže stíhá jízdu odpovídající zpracovávané události, vezmeme ho z patřičné výstupní fronty. Pokud nestíhá, musíme přidat nového aktivního dobrovolníka.

Časová i paměťová složitost tohoto řešení je $O(N)$, kde N je celkový počet jízd.

Pomalejší řešení

Můžeme postupovat také jinak. Vezmeme prvního dobrovolníka, pošleme ho na úplně první jízdu, potom na nejbližší další takovou jízdu, kterou stíhá, atd. Jestliže nám zbyly neuskutečněné jízdy, přidáme druhého člověka, pak případně ještě třetího, atd., dokud nepokryjeme všechny jízdy.

Důkaz správnosti tohoto řešení vypadá podobně jako v případě našeho vzorového řešení. (Přesněji, kdybychom do našeho vzorového řešení přidali podmínku „máš-li k dispozici více aktivních dobrovolníků, pošli na tobogan toho s nejmenším pořadovým číslem“, sestrojili bychom totéž řešení, jako tímto postupem.)

Přímočará implementace má v nejhorším možném případě časovou složitost $O(N^2)$. Existuje i implementace tohoto řešení s časovou složitostí $O(N \log N)$.

Za zmínku stojí také pomalejší řešení, které je ale obecnější a dalo by se použít i tehdy, kdyby jednotlivé tobogany začínaly a končily na různých místech v aquaparku, takže pěší přesuny mezi jejich konci a začátky by trvaly různě dlouho. Úlohu převedeme na hledání maximálního párování v bipartitním grafu. Jednu část grafu budou tvořit časy konců jízd, druhou časy začátků. Hrana znamená, že se po daném konci jízdy stíhá daný začátek další jízdy. Každému platnému rozvrhu jízd odpovídá nějaké párování v tomto grafu a naopak. Každá hrana zařazená do párování vlastně znamená, že člověk, který právě dokončil jednu jízdu, má jako následující uskutečnit příslušnou druhou jízdu. Je zjevné, že každou hranou zařazenou do párování ušetříme jednoho člověka. Hledaný minimální počet lidí proto můžeme spočítat jako N minus velikost maximálního párování v našem bipartitním grafu.

```
#include <cstdio>
#include <queue>

using namespace std;

#define MAX_TIME 2000000100
```

```
int main()
{
	int t[3], d, n[3], a;
	queue<int> nastup[3];
	queue<int> vystup[3];
	scanf("%d%d%d%d", &t[0], &t[1], &t[2], &d);
	for (int i = 0; i < 3; i++)
	{
		scanf("%d", &n[i]);
		for (int j = 0; j < n[i]; j++)
		{
			scanf("%d", &a);
			nastup[i].push(a);
		}
		nastup[i].push(MAX_TIME); // vložíme fiktivní poslední nástup
	}
	int prvniNastupPos, prvniNastupCas;
	int prvniVystupCas, prvniVystupPos;
	int pocetDeti = 0;
	while(1)
	{
		prvniNastupCas = MAX_TIME;
		prvniNastupPos = -1;
		for (int i = 0; i < 3; i++) // najdeme první nezpracovaný nástup
		{
			if (nastup[i].front() < prvniNastupCas)
			{
				prvniNastupCas = nastup[i].front();
				prvniNastupPos = i;
			}
		}
		if (prvniNastupCas == MAX_TIME)
			break; // vyčerpali jsme všechno, končíme
		prvniVystupCas = MAX_TIME;
		for (int i = 0; i < 3; i++) // najdeme první nepoužitý výstup
		{
			if (vystup[i].size() == 0)
				continue;
			if (vystup[i].front() < prvniVystupCas)
			{
				prvniVystupCas = vystup[i].front();
				prvniVystupPos = i;
			}
		}
		if (prvniVystupCas <= prvniNastupCas) // můžeme tento výstup použít
			vystup[prvniVystupPos].pop();
		else
			pocetDeti++;
		nastup[prvniNastupPos].pop();
		vystup[prvniNastupPos].push(prvniNastupCas + t[prvniNastupPos] + d);
	}
	printf("%d\n", pocetDeti);
	return 0;
}
```

P-II-2 Oplocení farmy

Vzorové řešení tohoto příkladu bude využívat stejný postup, jaký jsme použili v úloze o čokoládě z domácího kola. Každý čtvercový úsek farmy můžeme jednoznačně identifikovat bodem, v němž leží jeho pravý dolní roh, a délkou jeho strany. Základní myšlenkou řešení je, že místo počítání pestrých (tzn. vícebarevných) čtverců můžeme spočítat všechny čtverce a od jejich počtu odečíst počet jednobarevných čtverců.

Kdybychom pro každý bod (r, s) znali počet jednobarevných čtverců, jejichž pravý dolní roh je (r, s), dokázali bychom pak už hledaný výsledek snadno spočítat. Počet všech čtverců (jednobarevných i pestrých dohromady) s pravým dolním rohem (r, s) je totiž $\min(r, s)$. Počet pestrých čtverců pro (r, s) spočítáme tedy jako $\min(r, s)$ minus počet jednobarevných čtverců pro (r, s).

Podívejme se pozorněji na čtverce různých velikostí, které mají pravý dolní roh na políčku (r, s). Je-li některý z nich pestrý, jsou pestré i všechny čtverce větší, neboť ty ho celý obsahují.

Označme $K(r, s)$ délku strany největšího čtverce, který má pravý dolní roh na (r, s) a je jednobarevný (všechny plodiny v něm jsou stejné). Číslo $K(r, s)$ je zároveň rovno počtu všech jednobarevných čtverců, jejichž pravý dolní roh je (r, s).

Jak spočítáme hodnotu $K(r, s)$? Uvažujme políčka $(r, s-1)$, $(r-1, s)$ a $(r-1, s-1)$. Je-li aspoň na jednom z těchto políček jiná plodina než na políčku (r, s), potom je $K(r, s)$ rovno jedné, jelikož čtverec s délkou strany dva už je pestrý. V případě, že se plodiny pěstované na těchto třech políčkách shodují s plodinou na políčku (r, s), potom platí vztah:

$$K(r, s) = 1 + \min(\, K(r-1, s),\, K(r, s-1),\, K(r-1, s-1)\,).$$

Důkaz této rovnosti je stejný jako důkaz uvedený v řešení úloh domácího kola.

Protože k výpočtu $K(r, s)$ stačí znát $K(r-1, s)$, $K(r, s-1)$, $K(r-1, s-1)$, můžeme postupovat po řádcích shora dolů a v rámci každého řádku zleva doprava. Každou hodnotu přitom získáme v konstantním čase s využitím hodnot, které už známe. Řešení má proto časovou složitost $O(RS)$. Kdybychom si načetli a uložili celý vstup, měli bychom také paměťovou složitost $O(RS)$. Podobně jako v domácím kole můžeme paměťovou složitost snížit na $O(R)$, když si uvědomíme, že stačí udržovat v paměti pouze dva řádky vstupu a dva řádky hodnot $K(r, s)$, jelikož předcházející řádky již nebudeme potřebovat.

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

int R, S, K;
int M[2][2500], D[2][2550];
long long int res;

int min(int a, int b) { return a < b ? a : b; }
```

```
int main()
{
        scanf("%d%d%d", &R, &S, &K);
        // úplně nahoře a úplně vlevo si domyslíme imaginární políčka
        // s plodinou číslo K, aby se nám snadněji počítalo
        for (int i=0; i<=S; i++) D[0][i] = K;
        for (int i=1; i<=R; i++)
        {
                int novy = i % 2;
                int stary = (novy+1) % 2;
                D[novy][0] = K;
                for (int j=1; j<=S; j++) scanf("%d", &D[novy][j]);
                for (int j=1; j<=S; j++)
                {
                        if (D[novy][j-1] != D[novy][j] ||
                            D[stary][j] != D[novy][j] ||
                            D[stary][j-1] != D[novy][j])
                                M[novy][j] = 1;
                        else
                                M[novy][j] = 1 + min( min(M[novy][j-1], M[stary][j]),
                                                      M[stary][j-1] );
                        res += min(i, j) - M[novy][j];
                }
        }
        printf("%lld\n", res);
        return 0;
}
```

P-II-3 Omezovač rychlosti

Ať si zvolíme jakoukoliv trasu, optimální nastavení omezovače rychlosti bude jistě rovno minimu z maximální rychlosti povolené na cestách tvořících naši trasu – méně se nevyplatí, více nesmíme. Pro dosažení nejnižší doby jízdy mezi dvěma městy má proto smysl nastavovat omezovač rychlosti jedině na některou z rychlostí, které jsou zadány na vstupu.

Pomalejší řešení

Uvažujme nejprve, za jaký čas se dokážeme dostat z města x do města y, když už máme nastaveno omezení na nějakou rychlost v. Nech G_v je podgraf původního grafu, který obsahuje jen hrany, na nichž je rychlost v ještě povolena. Máme-li tedy nastaveno omezení na rychlost v, můžeme jet pouze po hranách grafu G_v. Jelikož rychlost jízdy už máme určenou, nejlepší dobu jízdy dosáhneme tehdy, pokud zvolíme nejkratší možnou trasu (v kilometrech).

Tím dostáváme následující řešení: Pro každou rychlost v ze vstupu vytvoříme graf G_v. V něm určíme délky (v kilometrech) nejkratších cest mezi všemi dvojicemi měst x a y – označíme je $dist_v(x, y)$. Výsledné optimální doby jízdy mají hodnotu $dist_v(x, y)/v$.

Jakou časovou složitost má toto řešení? Podle našeho úvodního pozorování stačí jako hodnoty v uvažovat všechny rychlosti zadané na vstupu – těch je celkem M. Graf G_v sestrojíme snadno v čase $O(M)$ tak, že projdeme všechny hrany původního

grafu. Zbývá vyřešit standardní problém určení délky nejkratší cesty v grafu G_v mezi každou dvojicí vrcholů. Nabízejí se dva možné přístupy:

- Floydův-Warshallův algoritmus, který tento problém řeší v čase $O(N^3)$.
- N-krát spuštěný Dijkstrův algoritmus (pro každý výchozí vrchol). Tím dojdeme opět k časové složitosti $O(N^3)$, nebo v případě využití haldy získáme řešení v čase $O(NM \log N)$.

Jelikož celý výpočet se provádí pro každou rychlost ze vstupu (tj. nejvýše M-krát), celková časová složitost popsaného řešení je $O(MN^3)$, příp. $O(M^2 N \log N)$, a paměťová složitost $O(N^2)$.

Vzorové řešení

Naše vzorové řešení využívá podobnou myšlenku jako Floydův-Warshallův algoritmus. Nejprve seřadíme všechny hrany ze vstupu do posloupnosti $e_1, e_2, \ldots, e_m$ tak, aby pro jejich maximální povolené rychlosti platilo $v_1 \geq v_2 \geq v_3 \geq \ldots \geq v_m$. Úlohu nyní vyřešíme metodou dynamického programování. Postupně pro každé k spočítáme všechny hodnoty $d_k(x, y)$ – délka nejkratší cesty (v kilometrech) mezi městy x a y, jestliže se smíme pohybovat pouze po hranách $e_1, e_2, \ldots, e_k$ (resp. $d_k(x, y) = \infty$, pokud taková cesta neexistuje.) Podle definice z předcházejícího řešení můžeme také říci, že $d_k(x, y)$ je délka nejkratší cesty mezi městy x a y v grafu G_{v_k}. Jak jsme již ukázali, k vyřešení úlohy stačí nalézt hodnoty $d_k(x, y)$ pro každé $k = 1, 2, \ldots, M$.

V k-tém kroku výpočtu chceme "zpřístupnit" hranu e_k a přepočítat hodnoty $d_k(x, y)$ pro všechny dvojice měst x, y. Při stanovení hodnoty $d_k(x, y)$ uvažujeme dvě možnosti.

- V případě, že nejkratší cesta z města x do města y nepoužije hranu e_k, je $d_k(x, y) = d_{k-1}(x, y)$.
- V opačném případě lze nejkratší cestu z x do y rozdělit na tři úseky: z vrcholu x půjdeme do některého vrcholu hrany e_k, projdeme touto hranou a z jejího druhého konce půjdeme do vrcholu y. V prvním, i ve třetím úseku cesty se vyskytují pouze hrany s číslem menším než k, takže optimální délku těchto úseků už známe.

  Jestliže označíme $u_{k,1}$ a $u_{k,2}$ vrcholy spojené hranou e_k, potom platí buď

  $$d_k(x, y) = d_{k-1}(x, u_{k,1}) + |e_k| + d_{k-1}(u_{k,2}, y)$$

  nebo

  $$d_k(x, y) = d_{k-1}(x, u_{k,2}) + |e_k| + d_{k-1}(u_{k,1}, y)$$

  podle toho, kterým směrem naše trasa prochází hranou e_k.

Můžeme si zvolit, zda hranu e_k použijeme nebo ne, a pokud ano, tak v kterém směru. Vybereme si samozřejmě nejlepší z uvedených tří možností. Proto je hodnota $d_k(x, y)$ rovna minimu z uvedených tří možností.

Tím jsme získali rekurzivní vztah pro výpočet hodnot $d_k(x, y)$. Každou z těchto MN^2 hodnot pomocí něho spočítáme v konstantním čase, celé řešení má proto časovou složitost $O(MN^2)$.

Pro snížení paměťové složitosti si stačí uvědomit, že hodnoty d_k závisejí jenom na hodnotách d_{k-1}. Vystačíme proto s pamětí velikosti $O(N^2)$.

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

struct hrana
{
      double m, d;
      int x, y;
};

int operator < (const hrana &h1, const hrana &h2) { return h1.m > h2.m; }

const double INF = 1e50;
const int maxM = 1000, maxN = 50;
int n, m;
hrana h[maxM];
double vysl[maxN][maxN];      // celkový výsledek, tj. časy mezi městy
double d[maxN][maxN];         // vzdálenost při použití prvních k-1 hran

int main()
{
      scanf("%d%d", &n, &m);
      for (int k=0; k<m; k++)
      {
            scanf("%d%d%lf%lf", &h[k].x, &h[k].y, &h[k].d, &h[k].m);
            h[k].x--; h[k].y--;
      }
      sort(h, h+m*sizeof(hrana));
      for (int i=0; i<n; i++)
            for (int j=0;j<n;j++)
                  d[i][j] = vysl[i][j] = (i==j) ? 0.0 : INF;

      for (int k=0; k<m; k++)
      {
            double v = h[k].m;
            for (int i=0; i<n; i++)
                  for (int j=0; j<n; j++)
                  {
                        d[i][j] = min(d[i][j], d[i][h[k].x] + h[k].d + d[h[k].y][j]);
                        d[i][j] = min(d[i][j], d[i][h[k].y] + h[k].d + d[h[k].x][j]);
                        vysl[i][j] = min(vysl[i][j], d[i][j]/v);
                  }
      }
      for (int i=0; i<n; i++)
      {
            for (int j=0; j<n; j++)
                  printf("%.3lf ", vysl[i][j]);
            printf("\n");
      }
      return 0;
}
```

P-II-4 Počítač Kvak

Část a: Lucasova čísla

(V celém řešení automaticky předpokládáme, že všechna sčítání se počítají modulo 65 536.)

Použijeme dva registry a a b. Na začátku výpočtu do nich vložíme hodnoty $L_0 = 2$ a $L_1 = 1$.

Představme si situaci, že v a máme hodnotu L_{x-2}, v b hodnotu L_{x-1} a roura je prázdná. Hodnotu L_x spočítáme tak, že vložíme do roury hodnoty z registru a a z registru b a provedeme příkaz `add`. Následně do roury vložíme ještě jednou hodnotu z registru b. V rouře tedy budeme mít za sebou nejprve hodnotu $L_{x-2} + L_{x-1} = L_x$ a potom hodnotu L_{x-1}. První z nich přečteme do registru b a druhou do registru a. Tím jsme se posunuli o jeden krok výpočtu: začínali jsme s hodnotami L_{x-2} a L_{x-1}, skončili jsme s hodnotami L_{x-1} a L_x.

Nyní již dokážeme snadno napsat celý program řešící zadanou úlohu: Číslo z roury uložíme do registru n. Inicializujeme registry a a b na hodnoty L_0 a L_1. Výše uvedený postup n-krát zopakujeme, čímž dostaneme v registrech a a b hodnoty L_n a L_{n+1}. Nakonec hodnotu z registru a vypíšeme na výstup.

```
get n
put 2 ; get a
put 1 ; get b
label loop
jz n konec
put n ; put 1 ; sub ; get n
put a ; put b ; add
put b
get b
get a
jump loop
label konec
put a
print
```

Část b: hledání výskytu čísla 47

K otestování, zda je číslo rovno 47, použijeme příkaz `jeq` – skok v případě rovnosti hodnot dvou registrů. Nejprve ale musíme dostat hodnotu 47 do některého registru. To nemůžeme provést jednoduše tak, že ji vložíme do roury a hned přečteme do registru, neboť rouru máme na začátku výpočtu plnou vstupních údajů. Celý obsah roury proto budeme muset přetočit.

Do roury nejprve vložíme hodnotu 0, která bude sloužit jako zarážka za vstupní posloupností kladných čísel. Za tuto 0 vložíme číslo 47. Nyní budeme opakovaně číst čísla z roury do registru a vkládat je zpět do roury, dokud nepřečteme naši nulu. V tomto okamžiku víme, že prvním číslem v rouře je 47 a za ním následuje celá původní vstupní posloupnost. Hodnotu 47 si tedy přečteme do registru z. Tím se dostaneme do stejné situace, jako na začátku výpočtu, jen s jediným rozdílem –

v registru z máme připraveno číslo 47. S ním teď každou hodnotu z roury porovnáme. Pokud se nám roura vyprázdní, aniž bychom v ní hodnotu 47 našli, vložíme do roury 0, vypíšeme ji na výstup a skončíme. Když naopak hodnotu 47 najdeme, nejprve v cyklu vyprázdníme zbytek roury, pak do ní vložíme hodnotu 1, vypíšeme ji na výstup a skončíme.

```
put 0
put 47
label pretoc
get a ; jz a dal ; put a
jump pretoc

label dal
get z

label testuj
jempty nenasel
get a
jeq a z nasel
jump testuj

label nasel
jempty pis1
get x
jump nasel

label pis1
put 1 ; print
stop

label nenasel
put 0 ; print
```

Pro zajímavost uvedeme ještě jednu možnost, jak lze řešit druhou část úlohy. Za posloupnost si opět vložíme nulu jako zarážku. Postupně čteme zadanou posloupnost a vždy, když narazíme na hodnotu 47, vložíme do roury číslo 1. Po přečtení nuly-zarážky, vložíme do roury ještě jednu 0.

V tomto okamžiku tedy máme v rouře nejprve tolik jedniček, kolikrát byla ve vstupní posloupnosti obsažena hodnota 47, a za nimi jednu nulu. První číslo z roury vypíšeme na výstup a skončíme.

Část c: výpis sudých čísel

Pro zjištění, zda je nějaké číslo sudé, spočítáme zbytek čísla po dělení 2 a ověříme, zde je roven nule. Zbytek po celočíselném dělení umíme určit příkazem `mod`. Zároveň si ale musíme někde uchovat i původní hodnotu čísla, neboť příkaz `mod` svoje vstupy z roury odstraní.

V první fázi řešení si upravíme obsah roury tak, abychom mohli následně používat příkaz `mod` na výpočet zbytků po dělení dvěma. Použijeme stejný trik jako v předchozí úloze – začneme tím, že za vstupní posloupnost vložíme 0 jako zarážku.

Nyní budeme po jednom číst čísla ze vstupní posloupnosti a za každé přečtené číslo a do roury postupně vložíme hodnoty 1, a, 2, a. Hodnota 1 bude signalizovat, že ještě následuje další číslo, další dvě hodnoty a a 2 použijeme jako vstupy pro příkaz `mod` a poslední hodnota a zůstane zachována.

Druhou fázi řešení zahájíme opět vložením zarážky 0 na konec posloupnosti. Potom opakovaně čteme čísla z roury do zvoleného registru. Když přečteme nulu, máme zpracovanou celou posloupnost a přejdeme na třetí fázi výpočtu. Jinak provedeme příkaz `mod`, který vyzvedne ze začátku roury hodnoty a a 2 a místo nich vloží na konec roury a mod 2. Následně přečteme z roury hodnotu a do registru a vložíme ji zpět do roury.

Jestliže v rouře byla původně posloupnost čísel $(s_1, \ldots, s_k)$, pak po skončení druhé fáze našeho řešení tam budeme mít posloupnost $(s_1 \bmod 2, s_1, s_2 \bmod 2, s_2, \ldots, s_k \bmod 2, s_k)$.

Ve třetí, závěrečné fázi výpočtu použijeme spočítané zbytky po dělení dvěma k tomu, abychom určili, která čísla máme vypsat a která nikoli. Přečteme vždy zbytek, je-li roven 1, následující číslo zahodíme, je-li roven 0, následující číslo vypíšeme na výstup. Po vyprázdnění roury výpočet skončí.

```
put 0
label mark
get a ; jz a druha_faze
put 1 ; put a ; put 2 ; put a
jump mark

label druha_faze
put 0
label dale
get a ; jz a vypis
mod
get a ; put a
jump dale

label vypis
jempty konec
get b
jz b chceme
get a
jump vypis

label chceme
print
jump vypis

label konec
```