## P-III-1 Hřiště

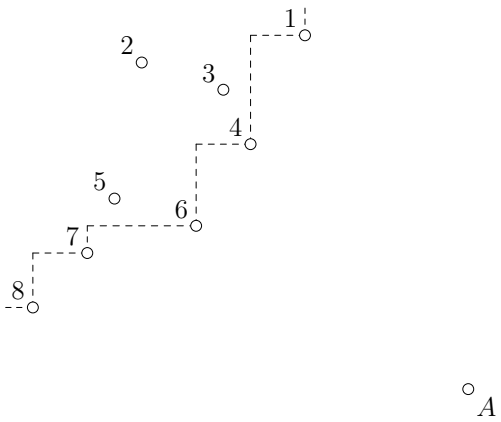
Řešení v kubickém čase je přímočaré: pro každou dvojici bodů se podíváme na jimi určený obdélník a pro každý ze zbývajících $n-2$ bodů ověříme, zda v něm leží. Rychlejší řešení už takto jednoduchá nejsou.

**Řešení v kvadratickém čase**

Každý obdélník má bod buď v levém dolním, nebo v pravém dolním rohu. Nám bude stačit spočítat ty, které mají bod v pravém dolním rohu. Když najdeme vhodný algoritmus, použijeme ho dvakrát – jednou přímo na zadaný vstup, potom rovinu překlopíme (změníme znaménko $x$-ovým souřadnicím všech bodů) a použijeme náš algoritmus podruhé.

V tomto řešení budeme zkoumat každý možný pravý dolní roh zvlášť. Máme tedy konkrétní bod $A$ a ptáme se, kolik různých prázdných obdélníků v něm má svůj pravý dolní roh. Jako levý horní roh lze použít samozřejmě pouze body, které vzhledem k $A$ leží ve druhém kvadrantu – tj. vlevo nahoře od bodu $A$.

Uvažujme tedy tuto množinu bodů. Máme-li v této množině dva body $X$ a $Y$ takové, že $Y$ je v obou souřadnicích blíže k $A$ než $X$, řekneme, že bod $Y$ zakrývá bod $X$. Například na následujícím obrázku vidíme, že bod 4 zakrývá body 2 a 3. Levými horními rohy prázdných obdélníků jsou potom zjevně právě ty body, které nejsou zakryty žádným jiným bodem.

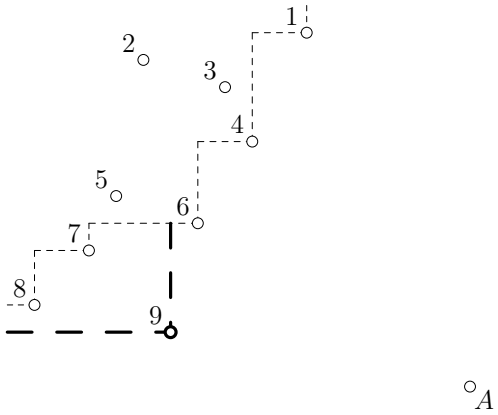



*Příklad rozmístění bodů vlevo nahoře od bodu A. Čárkovaná čára představuje hranici oblasti, kterou tyto body zakrývají. Každý bod, který leží na hranici, je levým horním rohem prázdného obdélníka. A naopak, každý jiný bod má mezi sebou a bodem A alespoň jeden z bodů ležících na hranici.*

Body si můžeme uspořádat shora dolů a v tomto pořadí je zpracovávat. Průběžně si budeme pamatovat množinu bodů, které aktuálně určují prázdné obdélníky. Tyto body nazveme kandidáti. Vždy po zpracování nového bodu odstraníme ty kandidáty, které tento nový bod „zakryl“.

Množinu aktuálních kandidátů si budeme pamatovat v obyčejném zásobníku, přičemž na vrcholu zásobníku bude naposledy zpracovaný kandidát. Všimněte si, že když projdeme zásobníkem zdola nahoru (od nejdříve vložených prvků – na obrázku v pořadí 1, 4, 6, 7, 8), obě jejich souřadnice klesají.

Při zpracování nového bodu vždy víme, že má menší $y$-ovou souřadnici než všechny dříve zpracované body. Zakryje tedy všechny kandidáty, kteří mají větší $x$-ovou souřadnici – a ty máme právě na vrcholu zásobníku, takže stačí postupně je ze zásobníku odebírat a zahazovat. Vypadá to například takto:




*Příklad zpracování dalšího bodu. Nový bod 9 leží níže než všichni dosavadní kandidáti. Po přidání bodu 9 postupně zjistíme, že zakryl kandidáty 8 a 7, ty tedy ze zásobníku odstraníme. Další kandidát 6 už je více vpravo než bod 9, takže přestaneme odstraňovat. Nakonec přidáme bod 9 na vrchol zásobníku s kandidáty.*

Kdybychom provedli tento postup zvlášť pro každý bod $A$, dostali bychom řešení s časovou složitostí $\mathcal{O}(n^2 \log n)$: pro každý z $n$ bodů projdeme všechny ostatní, vybereme mezi nimi body ležící vlevo nahoře od něj, uspořádáme je a následně je zpracujeme zleva doprava výše uvedeným postupem.

Výběr i zpracování dokážeme vykonat v lineárním čase. Při zpracování si všimněte, že každý bod jednou přidáme mezi kandidáty a nejvýše jednou ho odtud odebereme, takže celkově celé zpracování změní množinu kandidátů nejvýše $2n$-krát.

Nejpomalejší částí popsaného algoritmu je třídění. Stačí si však uvědomit, že pořadí bodů se mezi jednotlivými kroky výpočtu nemění. Stačí tedy pouze jednou na začátku v zanedbatelném čase $\mathcal{O}(n \log n)$ uspořádat všech $n$ bodů. Až potom budeme zpracovávat konkrétní bod $A$, projdeme tento uspořádaný seznam bodů a přeskočíme v něm bod $A$ a také všechny body, které neleží vlevo nahoře od $A$. Takto zpracujeme

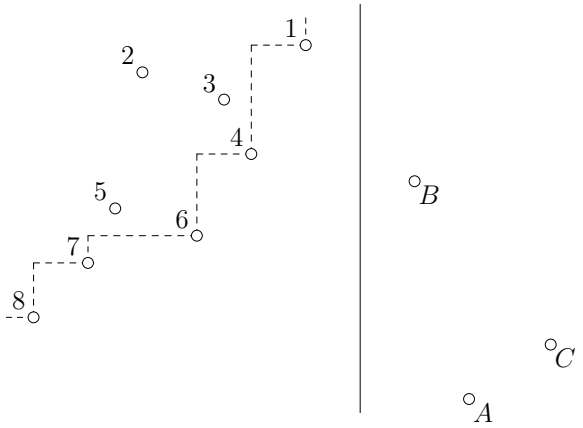
každý konkrétní bod $A$ v lineárním čase, takže dostáváme časovou složitost celého řešení $\mathcal{O}(n^2)$.

**Vzorové řešení**

Ve vzorovém řešení využijeme myšlenky kvadratického řešení a přidáme k nim ještě něco navíc. Také nyní budeme počítat pouze obdélníky, které mají body vlevo nahoře a vpravo dole.

Prvním vylepšením bude využití techniky *rozděl a panuj*. Máme-li jenom jeden bod, řešení úlohy je triviální. Když jich máme více, rozdělíme je svislou čarou na dvě pokud možno stejné poloviny: levou a pravou. Prázdné obdélníky jsou nyní tří typů: obdélníky ležící celé v levé polovině (jejich počet zjistíme rekurzivním voláním pro levou polovinu bodů), ležící celé vpravo (druhé rekurzivní volání) a takové, které mají svůj levý horní roh v levé polovině a svůj pravý dolní roh v pravé polovině.

Potřebujeme ještě umět efektivně spočítat prázdné obdélníky třetího typu. Zde je znázorněn příklad, jak může situace vypadat:




*Příklad situace při hledání obdélníků mezi levou a pravou polovinou. Plná svislá čára představuje hranici mezi polovinami. Už víme, že k bodu $A$ musí být levé horní rohy v aktuální množině kandidátů (na čárkované čáře). Nyní si navíc musíme všimnout, že nahoru může náš obdélník zasahovat nejvýše po bod $B$, takže pouze spodní tři kandidáti (8, 7, 6) skutečně určují prázdné obdélníky.*

Pro každý bod $A$ v pravé polovině chceme zjistit, kolik různých obdélníků určuje. Nejprve potřebujeme pro každý takový bod zjistit, kde je „jeho bod $B$“ – tedy nejbližší vyšší z bodů pravé poloviny, který leží nalevo od $A$. Tento bod představuje hranici, kam až nahoru může zasahovat obdélník s rohem v $A$.

Výpočet této informace je v podstatě totožný s algoritmem, jímž jsme v předchozím řešení hledali množinu kandidátů – jenom ho provádíme zdola nahoru a vždy, když přidáváme nového kandidáta, tento kandidát bude hranicí pro body, které v tu chvíli ze seznamu kandidátů odstraníme.

Pro řádově $n$ bodů toto můžeme provést v čase $\mathcal{O}(n \log n)$ kvůli třídění. Protože i poslední část tohoto řešení bude mít stejnou časovou složitost, toto nám bude stačit.

Zbytek řešení je založen na následující hlavní myšlence: Všechny body (levou i pravou polovinu dohromady) uspořádáme shora dolů a v tomto pořadí je budeme zpracovávat. Na levé straně si budeme v zásobníku udržovat množinu kandidátů. Kdykoliv přijde na řadu bod z levé poloviny, tuto množinu upravíme (stejně jako v předchozím řešení). Kdykoliv přijde na řadu bod z pravé poloviny, podíváme se na horní hranici pro jeho obdélníky a tuto souřadnici binárně vyhledáme v naší množině kandidátů vlevo. Celkový počet prázdných obdélníků zvýšíme o počet kandidátů, kteří leží dostatečně nízko – každý z nich určuje jeden prázdný obdélník.

Zbývá odvodit časovou složitost popsaného algoritmu. Označíme $T(n)$ celkový čas potřebný na zpracování $n$ bodů. Toto zpracování se skládá ze dvou rekurzivních volání na zpracování $n/2$ bodů, následného předpočítání horních hranic pro pravou polovinu a potom sčítání obdélníků, které zasahují do obou polovin. Každé rekurzivní volání proběhne v čase $T(n/2)$ a zbytek výpočtu dokážeme provést v čase $\Theta(n \log n)$. Dostáváme tedy, že funkce $T$ splňuje následující rekurzivní vztah: $T(n) = 2T(n/2) + \Theta(n \log n)$.

Toto je podobná rekurze, jako například při třídění MergeSort, jenom tam máme jeden logaritmus navíc. Podobnou technikou jako při analýze MergeSortu lze ukázat, že funkce $T$ patří do třídy $\Theta(n(\log n)^2)$, což je tedy i celková časová složitost našeho řešení.*

```
#include <bits/stdc++.h>
using namespace std;

struct point { int id, x, y; };

bool compare_X(const point &A, const point &B) { return A.x < B.x; }
bool compare_Y(const point &A, const point &B) { return A.y > B.y; }

ostream& operator<<(ostream &os, const point &P) {
    return os << "(" << P.x << "," << P.y << ")";
}

long long count(vector<point> body) {
    int N = body.size();
    if (N <= 1) return 0;

    // Rozdělíme body na levou a pravou polovinu, provedeme rekurzivní volání
    sort( body.begin(), body.end(), compare_X );
    vector<point> leve( body.begin(), body.begin()+(N/2) );
    vector<point> prave( body.begin()+(N/2), body.end() );
```

* Jeden možný argument vypadá následovně: Představte si všechna rekurzivní volání našeho algoritmu znázorněná jako strom. Celkový čas výpočtu algoritmu můžeme zjistit tak, že sečteme počty kroků, které vykoná v každém z vrcholů tohoto stromu. Toto sčítání provedeme po vrstvách: nejprve kořen, potom oba jeho syny, pak dohromady všechny čtyři vrcholy, v nichž zpracováváme $n/4$ bodů, a tak dále. Strom rekurze má zhruba $\log_2 n$ vrstev a na každé se celkem provede nejvýše řádově $n \log n$ kroků výpočtu.

```
    int hranice = prave[0].x;
    long long answer = count(leve) + count(prave);

    // Přepočítáme pro každý bod v pravé části, kam až nahoru můžeme
    sort( body.begin(), body.end(), compare_Y );
    vector<int> max_y(N, 1<<30);
    vector<int> visible;
    for (int n=N-1; n>=0; --n) {
        if (body[n].x < hranice) continue;
        while (!visible.empty() && body[n].x < body[visible.back()].x) {
            int b = visible.back();
            visible.pop_back();
            max_y[b] = body[n].y;
        }
        visible.push_back(n);
    }

    // Počítáme obdélníky
    visible.clear();
    for (int n=0; n<N; ++n) {
        if (body[n].x < hranice) {
            // bod z levé poloviny
            while (!visible.empty() && body[n].x > body[visible.back()].x)
                visible.pop_back();
            visible.push_back(n);
        } else {
            // bod z pravé poloviny
            int top = max_y[n];
            int lo = -1, hi = visible.size();
            while (hi - lo > 1) {
                int med = (lo + hi) / 2;
                if (body[visible[med]].y < top)
                    hi = med;
                else
                    lo = med;
            }
            answer += visible.size() - hi;
        }
    }
    return answer;
}

int main() {
    int N; cin >> N;
    vector<point> body(N);
    for (int n=0; n<N; ++n) cin >> body[n].x >> body[n].y;
    long long total = count(body);
    for (int n=0; n<N; ++n) body[n].x *= -1;
    total += count(body);
    cout << total << endl;
}
```

## P-III-2 Hořící strom

Pokud je strom malý a větve krátké, můžeme celou úlohu vyřešit pomocí dvou vnořených prohledávání. Kromě zadaných vrcholů stromu si můžeme vytvořit vrcholy také podél každé větve tak, aby mezi sousedními vrcholy byla vždy vzdálenost jeden centimetr. Tak dostaneme nový strom s jednotkovými hranami. Šíření ohně v něm budeme simulovat prohledáváním do šířky. Po každé sekundě simulace (tzn. vždy, když následující vrchol čekající ve frontě na zpracování má větší vzdálenost od počátku, než naposledy zpracovaný vrchol) se můžeme podívat na dosud nehořící části stromu a pomocí druhého prohledávání (tentokrát je jedno, zda do šířky nebo do hloubky) spočítáme komponenty souvislosti – tedy jednotlivé nehořící oblasti.

Lepší řešení získáme, když v předchozím řešení nebudeme simulovat každou sekundu šíření ohně zvlášť. Během těch sekund, v nichž se oheň pouze šíří po větvích stromu, se nijak nemění počet nehořících oblastí. To může nastat jedině tehdy, když oheň dohoří do některého z původních vrcholů, nebo když se na některé větvi setkají dva protisměrně hořící ohně. Takové situace nazveme *událostí*.

Strom s $n$ vrcholy má $n-1$ hran, takže celkem v něm může nastat jenom $\mathcal{O}(n)$ událostí. Efektivnější proto bude simulovat šíření ohně tak, že postupně budeme zjišťovat, kdy a které události během požáru nastanou.

Takovou simulaci provedeme poměrně snadno v kvadratickém čase. Postupně budeme určovat všechny okamžiky, v nichž nějaká událost nastane. Máme-li situaci v nějakém čase $t_i$ (přesněji těsně po něm), projdeme všechny hořící hrany a o každé zjistíme, kdy na ní nastane nejbližší událost. Minimum z takto zjištěných časů bude časem $t_{i+1}$ následující události.

Těsně po každém z časů $t_i$ (včetně času $t_0 = 0$, tedy hned po začátku simulace) spustíme prohledávání, abychom spočítali aktuální počet nehořících oblastí. Každé takové prohledávání vykonáme v lineárním čase – stačí si postupně obarvovat dosud nehořící vrcholy, přičemž nehořící vrcholy spojené nehořící hranou patří do téže komponenty. Nakonec ještě připočítáme jednu oblast za každou hranu, jejíž oba konce už hoří, ale hrana ještě neshořela celá.

Toto řešení potřebuje $\mathcal{O}(n^2)$ času na samotnou simulaci hoření a během ní $\mathcal{O}(n)$-krát spustí prohledávání v čase $\mathcal{O}(n)$ na spočítání nehořících oblastí. Celková časová složitost je proto kvadratická: $\mathcal{O}(n^2)$.

### Vzorové řešení 1: simulace zepředu

V předchozím řešení zlepšíme dvě věci. Abychom dokázali rychleji zpracovávat události, budeme si budoucí události, o nichž už víme, udržovat v prioritní frontě uspořádané podle času, kdy nastanou. A abychom dokázali rychleji určit počet nehořících oblastí, budeme ho přímo přepočítávat vždy po zpracování nějaké události.

Začneme druhou uvedenou změnou. Když se setkají dva ohně někde na hraně, přestane existovat oblast mezi nimi, počet oblastí tedy klesne o jednu. Když dohoří oheň do některého dosud nehořícího vrcholu, počet oblastí se zvýší, protože do té doby souvislá oblast se rozpadne na několik menších. Musíme si ovšem dát pozor na správné zpracování situace, když více událostí nastane najednou na stejném místě.

Jestliže do vrcholu $v$ stupně $d$ zároveň dohoří oheň po $k$ různých hranách, rozpadne se původní jedna oblast na $d - k$ menších – jedna pro každou hranu, po níž se z $v$ začne oheň šířit dále.

Nyní se podíváme na efektivnější zpracování událostí. Každou událost popíšeme třemi údaji: čas, kdy nastane, její typ (zda jde o událost na hraně nebo ve vrcholu) a identifikátor objektu, kde nastane. Události budeme řadit v prioritní frontě postupně podle všech těchto parametrů. Potom dokážeme zároveň zpracovat všechny události, které nastanou na stejném místě.

**Vzorové řešení 2: simulace zezadu**

Místo toho, abychom ošetřovali události, které nastanou zároveň, můžeme jenom „naprázdno“ provést výše popsanou simulaci. Nebudeme se při ní zajímat o nehořící oblasti, pouze vygenerujeme uspořádaný seznam všech událostí, které během požáru nastanou. V tomto seznamu stačí mít zjednodušené události „začal hořet tento vrchol“ a „dohořela někde uvnitř tato hrana“.

Jakmile získáme seznam událostí, můžeme si jakoby celý proces pustit odzadu. Když se na film s požárem díváme odzadu, vidíme nové dva typy událostí: „vznikla nová oblast na hraně“ a „přestal hořet vrchol, čímž se několik oblastí spojilo dohromady“. Takové typy událostí umíme efektivně simulovat pomocí datové struktury Union-Find.* Pokaždé, když se během této simulace změní aktuální čas, stačí se podívat na aktuální počet komponent souvislosti.

Tento přístup k řešení má navíc tu výhodu, že ho v podstatě beze změny dokážeme zobecnit na libovolné grafy. Tam by předchozí způsob simulace nefungoval, neboť jenom lokálně neumíme určit, jak se změnil počet a tvar nehořících oblastí, když začal hořet nový vrchol.

Obě verze vzorového řešení mají časovou složitost $\mathcal{O}(n \log n)$. (Ve druhém vzorovém řešení je k tomu zapotřebí dostatečně efektivní implementace datové struktury Union-Find.)

```
#include <bits/stdc++.h>
using namespace std;

const int VRCHOL = 0, HRANA = 1, NEKONECNO = 987654321;

struct udalost { int cas, typ, id; };

bool operator < (const udalost &A, const udalost &B) {
    if (A.cas != B.cas) return A.cas < B.cas;
    if (A.typ != B.typ) return A.typ < B.typ;
    return A.id < B.id;
}

bool operator > (const udalost &A, const udalost &B) { return B<A; }

struct hrana { int k1, k2, d, z1, z2; };
// Konce, délka a časy, kdy konce začaly hořet
```

* Popis této struktury najdete například v řešení úlohy P-III-5 v 59. ročníku MO-P.

```
void zapal(hrana &H, int u, int cas) {
    if (H.k1 == u) H.z1 = cas; else H.z2 = cas;
}

bool hori_oba_konce(const hrana &H) {
    return (H.z1 < NEKONECNO) && (H.z2 < NEKONECNO);
}

int kdy_dohori(const hrana &H) {
    int prvni = min(H.z1, H.z2), druhy = max(H.z1, H.z2);
    return prvni + (H.d - (druhy-prvni)) / 2;
}

int opacny_vrchol(const hrana &H, int u) {
    return H.k1 + H.k2 - u;
}

int main() {
    // Načteme vstup
    int N, H; cin >> N >> H;
    vector<int> rodic(N,-1), delka(N,-1);
    for (int n=1; n<N; ++n) cin >> rodic[n];
    for (int n=1; n<N; ++n) cin >> delka[n];

    // Sestrojíme si graf
    vector<hrana> hrany(N-1);
    vector< vector<int> > graf(N);
    for (int n=0; n<N-1; ++n) {
        int u = n+1, v = rodic[n+1];
        hrany[n] = { u, v, delka[n+1], NEKONECNO, NEKONECNO };
        graf[u].push_back(n);
        graf[v].push_back(n);
    }

    // Nachystáme si datové struktury pro simulaci
    int aktualnich_komponent = 1;
    int nejvic_komponent = 1;
    int posledni_cas = 0;
    priority_queue< udalost, vector<udalost>, greater<udalost> > udalosti;
    vector<int> zacal_horet(N, NEKONECNO);

    // Zapálíme počáteční vrcholy
    while (H--) {
        int v; cin >> v;
        zacal_horet[v] = 0;
        aktualnich_komponent += int(graf[v].size()) - 1;
        for (int h : graf[v]) {
            zapal(hrany[h], v, 0);
            if (hori_oba_konce(hrany[h]))
                udalosti.push( { kdy_dohori(hrany[h]), HRANA, h } );
            else
                udalosti.push( { hrany[h].d, VRCHOL, opacny_vrchol(hrany[h],v) } );
        }
    }
    nejvic_komponent = max(nejvic_komponent, aktualnich_komponent);

    // Zpracováváme události
    while (!udalosti.empty()) {
        auto U = udalosti.top(); udalosti.pop();
```

```
        if (U.cas > posledni_cas) {
            nejvic_komponent = max(nejvic_komponent, aktualnich_komponent);
            posledni_cas = U.cas;
        }
        if (U.typ == VRCHOL) {
            // Jde-li o událost ve vrcholu, zkontrolujeme, zda opravdu nastala,
            // nebo začal hořet dřív
            if (U.cas > zacal_horet[U.id]) continue;
            // Pokud vrchol začal hořet teď z více směrů najednou, ubývají komponenty
            if (U.cas == zacal_horet[U.id]) { --aktualnich_komponent; continue; }
            // Pokud vrchol ještě nezačal hořet, zapálíme ho
            zacal_horet[U.id] = U.cas;
            aktualnich_komponent += int(graf[U.id].size()) - 2;
            // A zapálíme i všechny hrany vedoucí z něj
            for (int h : graf[U.id]) {
                zapal(hrany[h], U.id, U.cas);
                if (hori_oba_konce(hrany[h])) {
                    int t = kdy_dohori(hrany[h]);
                    if (t > U.cas) udalosti.push( { t, HRANA, h } );
                } else {
                    udalosti.push(
                      { U.cas+hrany[h].d, VRCHOL, opacny_vrchol(hrany[h],U.id) } );
                }
            }
        } else {
            --aktualnich_komponent;
        }
    }
    cout << nejvic_komponent << endl;
}
```

### P-III-3 Vozidlo zpracovává pole

V částech B a C budeme pro jednoduchost předpokládat, že makra `spoj`, `první` a `druhý` máme implementována tak, že výstupní lokalita může být shodná s některou vstupní – tedy že například máme-li v lokalitě `X` kód dvojice $(a, b)$, můžeme provést příkaz `první X X` a získat tak přímo v lokalitě `X` hodnotu $a$.

Makra, která tuto vlastnost nemají, můžeme do této podoby snadno upravit: nejprve uložíme výstup do pomocné lokality a pak na konci makra přepíšeme příslušnou vstupní lokalitu správnou výstupní hodnotou.

#### Část A: testování prvočíselnosti

V této úloze stačí vhodně použít cykly a existující makra. Ošetříme $a = 0$ a $a = 1$ jako speciální případy. Pro $a \geq 2$ potřebujeme ověřit, zda $a$ není dělitelné žádným z čísel od 2 do $a - 1$. Už máme makro pro dělení se zbytkem, potřebujeme ho jenom v cyklu použít vícekrát a pokaždé zkontrolovat jeho výstup.

```
MAKRO prvočíslo A B
            zapiš A [A-kopie]
            vynuluj [D]
            vynuluj [nula]

            stejné A [D] konec              # otestujeme zda A=0 nebo A=1
```

```
            přenes K J [D] -
            stejné A [D] konec
            přenes K J [D] -                # v D máme nyní 2
                                            # zkusíme, zda D dělí A

    cyklus: stejné A [D] našel              # žádné D < A nedělilo A
                                            # => A je prvočíslo
            zapiš [A-kopie] A
            vyděl A [D] [podíl] [zbytek]
            stejné [zbytek] [nula] konec    # našli jsme dělitele
                                            # => A není prvočíslo
            přenes K J [D] -                # jdeme na další D
            skoč cyklus

    našel:  přenes K J B -
    konec:  zapiš [A-kopie] A
END
```

**Část B: přidej na konec**

Při řešení těchto úloh nám pomůže, když si proměnnou obsahující posloupnost čísel představíme jako zásobník. Základní operace prováděné s posloupností jsou totiž velmi podobné právě zásobníkovým operacím: dokážeme se snadno podívat na první prvek posloupnosti, odebrat ho, nebo naopak na začátek posloupnosti přidat nový prvek. Na naši úlohu potom můžeme nahlížet jako na úlohu „napiš program, který přidá nové číslo na dno zásobníku“.

Se zásobníky tuto úlohu vyřešíme snadno. Posloupnost prvek po prvku přesuneme do druhého zásobníku, čímž obrátíme pořadí jejích prvků. Její původní konec je nyní na začátku. Nový prvek přidáme na vrchol zásobníku, potom všechny prvky přesuneme zpět a jsme hotoví.

V naší implementaci si nejprve připravíme makra `push` a `pop`, která budou pracovat s posloupností jako se zásobníkem: `push` na začátek posloupnosti přidá novou hodnotu, `pop` odebere z posloupnosti její první prvek do určené proměnné. Pokud se to nepodaří (posloupnost už byla prázdná), skočí na příslušné návěstí.

Pomocí těchto maker napíše samostatné makro `reverz`, které obrátí posloupnost, a s jeho pomocí potom definujeme výsledné makro `append`.

```
MAKRO push X prvek
            spoj prvek X X
            přenes K J X -
END

MAKRO pop X prvek nevyšlo
            přenes X J K nevyšlo   # jestliže X=0, snažíme se odebrat
                                   # prvek z prázdné posloupnosti
            první X prvek
            druhý X X
END

MAKRO reverz X Y
            vynuluj Y
    cyklus: pop X [prvek] konec    # odebereme prvek nebo zjistíme,
                                   # že už tam žádné nejsou
```

```
            push Y [prvek]
            skoč cyklus
    konec:  čekej
END

MAKRO append X prvek
            reverz X [Z]
            push [Z] prvek
            reverz [Z] X
END
```

**Část C: třídění**

Jednou možnou cestou k vyřešení této úlohy je vytvořit makra, která nám umožní pracovat s posloupností jako s plnohodnotným polem – tedy makro pro čtení prvku na konkrétním indexu a makro pro zápis na konkrétní index. Ani potom ještě nebude implementace žádného třídicího algoritmu přímočará. Doporučujeme použít algoritmus InsertSort nebo SelectSort. Implementovat rekurzivní třídění jako QuickSort nebo MergeSort je sice možné, ale velmi pracné. V našem vzorovém řešení půjdeme ještě snadnější cestou: budeme implementovat vhodně upravený BubbleSort.

V našem řešení se obejdeme bez funkcí na práci s polem. Místo toho si všimněte, že výměny sousedních prvků dokážeme provést i během obracení posloupnosti (provádění operace `reverz`). Uděláme to jednoduše: Namísto toho, abychom vždy vzali prvek z jedné posloupnosti a vložili ho do druhé, podíváme se pokaždé na první dva prvky a do druhé posloupnosti vložíme menší z nich.

Jeden takový průchod posloupností přesně odpovídá jednomu průchodu polem při BubbleSortu. (Tedy až na to, že nám celou posloupnost převrátí. To ale snadno napravíme druhým, tentokrát již obyčejným průchodem s převrácením posloupnosti.)

Když chceme uspořádat $n$-prvkovou posloupnost, stačí zopakovat výše popsaný postup alespoň $(n-1)$-krát. Na konci posloupnosti se nám tím postupně nahromadí největší prvky ve správném pořadí. Každý průchod posloupností je ponechá na místě a navíc k nim přidá největší z prvků, které ještě nebyly na správném místě.

```
MAKRO minimum X Y Z                 # do Z uložíme minimum z hodnot X a Y
            zapiš X [Xkopie]
            přenes [Xkopie] Y Z nevyšlo
            skoč konec
   nevyšlo: zapiš X Z
   konec:   čekej
END

MAKRO maximum X Y Z                 # do Z uložíme maximum z hodnot X a Y
            vynuluj Z
            přenes K X Z -
            přenes K Y Z -
            minimum X Y [menší]
            přenes Z [menší] K -
END
```

```
MAKRO reverzsort X Y
            pop X [stranou] konec  # první prvek pole odložíme stranou
    cyklus: pop X [prvek] hotovo   # odebereme další prvek,
                                   # pokud už není, skončíme
            minimum [stranou] [prvek] [menší]
            maximum [stranou] [prvek] [větší]
            push Y [menší]
            zapiš [větší] [stranou]
            skoč cyklus
    hotovo: push Y [stranou]        # vrátíme zpět do posloupnosti
                                    # právě odložený prvek
    konec:  čekej
END

MAKRO sort X
            délka X [D]
    cyklus: přenes [D] J K konec
            reverzsort X [Y]
            reverz [Y] X
            skoč cyklus
    konec:  čekej
END
```