P-II-1 Halušky

Jednoduché řešení za pár bodů je následující: Po projití každé stezky je Michal na nějakém místě (n možností) a má nějakou hmotnost ($c = \max c_i$ smysluplných možností, těžší Michal by už nemohl bezpečně projít žádnou stezku). Pro každou počáteční hmotnost Michala proto můžeme postupně prohledávat graf, jehož vrcholy tvoří všech možných nc stavů místo–hmotnost a jehož hrany odpovídají jednotlivým pohybům Michala po Tatrách.

Lepší řešení můžeme založit na dvou pozorováních. První pozorování je, že když má Michal víc možností, jak se dostat z jednoho místa na druhé, je lepší ta z nich, na které přibere méně. Pro každé místo nás tedy bude zajímat jen nejnižší hmotnost, s níž se tam Michal umí dostat. Druhé pozorování je, že pokud se Michal umí do místa $n-1$ (budeme jej nazývat *cíl*) dostat se startovní hmotností x, umí to i s každou startovní hmotností $x' \leq x$ – tedy tato vlastnost je monotónní. Proto na vyhledání největšího takového x můžeme použít binární vyhledávání.

Dohromady tato dvě pozorování dají řešení s časovou složitosti $\mathcal{O}(m \log n \log c)$: Budeme mít řádově $\log c$ iterací binárního vyhledávání hmotnosti x a v rámci každé z nich použijeme Dijkstrův algoritmus, abychom pro každé místo našli nejmenší hmotnost, s níž se tam Michal umí dostat. Detaily tohoto řešení neuvádíme, jsou velmi podobné detailům vzorového řešení, které si nyní ukážeme.

Vzorové řešení

Pokud se na celou věc podíváme od konce, nebudeme potřebovat binární vyhledávání. Trik spočívá v tom, že si maximální hmotnost, s níž se z daného místa umíme dostat do cíle, spočítáme pro každé místo, nejen pro místo 0. Tyto hodnoty se chovají podobně jako vzdálenosti. Označme jako $D[i]$ největší hmotnost, o níž již víme, že se s ní Michal z místa i umí dostat do cíle (tato hmotnost již obsahuje případné halušky z místa i). Na začátku máme $D[n-1] = \infty$ a $D[i] = -\infty$ pro všechna $i < n-1$.

Hodnoty D můžeme postupně zvyšovat: Podívejme se na nějakou stezku, která spojuje místa a a b a má maximální hmotnost c. Pokud již známe $D[b]$, tak víme, že chceme-li se dostat z místa a do cíle, jednou z možností je projít právě uvažovanou stezkou do místa b a odtud se dostat do cíle. Aby se nám to podařilo, musíme projít stezku (a přitom mít hmotnost nejvýše c), následně musíme sníst halušky v místě b a skončit s hmotností nejvýše $D[b]$. To znamená, že v místě a potřebujeme mít hmotnost maximálně $\min(c, D[b] - h_b)$. Nová hodnota $D[a]$ je proto maximum ze současné hodnoty $D[a]$ a právě vyzkoušené možnosti.

Nyní můžeme udělat stejnou úvahu jako u Dijkstrova algoritmu. Náš algoritmus bude probíhat v kolech. V každém kole se podíváme na místa, která ještě nejsou uzavřená, a to z nich, které má aktuálně nejvyšší hodnotu D, prohlásíme za uzavřené. (Toto můžeme udělat, jelikož se tato hodnota nemá jak změnit.) Jakmile jej

uzavřeme, projdeme všechny stezky, které z něj vedou, a pro každý ještě neuzavřený vrchol vyzkoušíme výše popsaným způsobem zlepšit jeho aktuální hodnotu D.

Tento algoritmus má tedy stejnou složitost (a také téměř stejnou implementaci) jako klasický Dijkstrův algoritmus pro hledání nejkratších cest: $\mathcal{O}(m \log n)$ časovou a $\mathcal{O}(m + n)$ paměťovou.

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

const int NEKONECNO = 987654321;

struct hrana { int kam, limit; };

int main() {
    // Načítáme vstup
    int N, M;
    cin >> N >> M;
    vector<int> h(N);
    for (int n = 0; n < N; ++n) cin >> h[n];
    vector< vector<hrana> > G(N);
    for (int m = 0; m < M; ++m) {
        int a, b, c;
        cin >> a >> b >> c;
        G[a].push_back( { b, c } );
        G[b].push_back( { a, c } );
    }

    // Spustíme Dijkstrův algoritmus "od konce"
    vector<int> D(N, -NEKONECNO);
    D[N-1] = +NEKONECNO;
    set< pair<int,int> > Q;
    for (int n = 0; n < N; ++n) Q.insert( { D[n], n } );
    while (!Q.empty()) {
        int kde = (--Q.end())->second;
        Q.erase( --Q.end() );
        for (auto e : G[kde]) {
            int pred = e.kam, maxw = e.limit;
            int nove = min( maxw, D[kde] - h[kde] );
            if (nove > D[pred]) {
                Q.erase( { D[pred], pred} );
                D[pred] = nove;
                Q.insert( { D[pred], pred } );
            }
        }
    }
    if (D[0] > 0) {
        cout << D[0] << endl;
    } else {
        cout << (-1) << endl;
    }
}
```

P-II-2 Červené jablíčko

Vašek může lhát jen jednou. Co se tedy stane, když se ho na stejnou otázku zeptáme dvakrát? Pokud v obou případech dostaneme stejnou odpověď, tak víme, že tato odpověď je skutečně správnou odpovědí na naši otázku. No a naopak, pokud jednou dostaneme odpověď ANO a jednou odpověď NE, víme, že při jedné z nich Vašek lhal. Nevíme sice ještě, jaká je skutečná odpověď na otázku, na niž jsme se právě dvakrát zeptali, ale zato víme, že odteď musí Vašek na všechny další otázky odpovídat pravdivě.

Základní korektní strategie tedy může vypadat například následovně: Pepa bude postupně jedno po druhém brát do ruky jednotlivá jablka, každé ukáže Vaškovi a dvakrát se zeptá, zda je červené. Pokud někdy dostane dvakrát ANO, má červené jablko. Pokud někdy dostane ANO a NE, nachytal Vaška při lhaní a odteď se na každé jablko (včetně toho, které drží v ruce) bude ptát jen jednou.

Kdyby například Vašek lhal hned ve své první odpovědi, po prvních dvou otázkách bude Pepa vědět, že lhal, ale ještě nebude nic vědět o jablkách. Následně by mu už stačilo 999 dalších otázek – pokud se postupně o 999 jablkách dozví, že jsou zelená, to poslední musí být červené.

Nejhorší možný případ pro tuto strategii je, že až poslední jablko, které Pepa vezme, bude červené, a až při předposledním jablku Vašek jednou zalže. Takto Pepa 999krát položí dvě otázky a na závěr potřebuje ještě jednu, aby zjistil, zda je předposlední jablko červené či zelené. Dohromady tedy položí 1999 otázek.

Binární vyhledávání

Pepa samozřejmě může používat mnohem efektivnější strategie. Kdyby Vašek nikdy nelhal, Pepovou nejlepší strategií by bylo použít *binární vyhledávání*. Při každé otázce by stačilo rozdělit jablka na dvě přibližně stejně velké hromádky, ukázat na jednu z nich a Vaška se zeptat, zda je v ní to červené. Každá otázka by vyloučila zhruba polovinu jablek, proto by pro 1000 jablek (ve skutečnosti až pro $2^{10} = 1024$ jablek) Pepovi stačilo 10 otázek.

Stejnou strategii můžeme použít i v naší úloze, jen samozřejmě přidáme techniku, již jsme objevili výše, totiž že dokud nevíme, zda již Vašek lhal, každou otázku pokládáme dvakrát. Nejhorším případem je zjevně situace, kdy Vašek bude lhát, až když Pepovi zůstanou poslední dvě jablka. Tehdy Pepa položí postupně 21 otázek: Prvních 20 na simulaci binárního vyhledávání a poslední na závěrečné rozlišení mezi posledními dvěma jablky.

Vzorové řešení

Ukážeme dva možné pohledy na vzorové řešení. V obou případech budeme úlohu řešit pro 1024 jablek namísto 1000, ušetříme si tak starosti se zaokrouhlováním. (Pepa si tedy kromě 1000 reálných jablek představí 24 virtuálních, o nichž ví, že nejsou červená, takže je jedno, zda je Vaškovi fyzicky ukazuje, anebo ne.)

První možný pohled na úlohu je takový, že nás pro každé jablko zajímá, kolikrát nám o něm Vašek tvrdil, že je zelené. Jakmile se to o nějakém jablku dozvíme dvakrát, víme, že je opravdu zelené a můžeme ho zahodit. Jablka tedy budeme dělit

na tři hromádky. Na první jsou ty, o nichž Vašek nikdy netvrdil, že jsou zelená (klidně i proto, že jsme se na ně ještě neptali), na druhé jsou ty, o nichž to tvrdil jednou a na třetí, odpadové, hromádce jsou jablka, o nichž Vašek dvakrát tvrdil, že jsou zelená, a tedy s jistotou zelená jsou.

Každý možný stav si můžeme popsat dvojicí čísel udávající počty jablek na první a druhé hromádce. Na začátku jsme ve stavu $(1024, 0)$. Nechť si nyní Pepa jako první otázku vybere x jablek a zeptá se, zda je červené mezi nimi. Pokud Vašek odpoví ANO, dostaneme se do situace $(x, 1024 - x)$ (když Vašek řekne, že v nějaké množině je červené jablko, tvrdí tím zároveň, že v jejím doplňku jsou samá zelená), pokud odpoví NE, dostaneme se do stavu $(1024 - x, x)$. Pokud rozdělíme jablka na dvě poloviny, bez ohledu na Vaškovu odpověď se dostaneme do stavu $(512, 512)$.

Pokračujme nyní v dělení jablek na poloviny, a to následovně. Při druhé otázce si vybereme 256 jablek z první a 256 jablek z druhé hromádky a zeptáme se Vaška, zda je to červené mezi nimi. Rozmyslíme si nyní, že bez ohledu na to, jak odpoví, se dostaneme do stavu $(256, 512)$: O 256 jablkách z druhé hromádky Vašek najisto prohlásí, že jsou zelená, takže je zahodíme, druhých 256 jablek z druhé hromádky na ní zůstane. Stejně tak na první hromádce zůstane 256 jablek a druhých 256 jablek přesuneme na druhou.

I v dalším kroku rozdělíme hromádky na poloviny a zopakujeme tutéž úvahu. Na první hromádce nám jich zůstane 128, na druhé 256 a navíc k nim přesuneme 128 z první hromádky. Budeme tedy ve stavu $(128, 384)$. Ve stejném duchu proběhne čtvrtá až desátá otázka. Postupně se dostaneme do stavů $(64, 256)$, $(32, 160)$, $(16, 96)$, $(8, 56)$, $(4, 32)$, $(2, 18)$ a po desáté otázce budeme ve stavu $(1, 10)$.

Máme tedy právě jedno jablko, které je červené, pokud Vašek nikdy nelhal, a přesně deset jablek, z nichž každé je červené, pokud Vašek už právě jednou lhal. Mezi těmito jedenácti jablky potřebujeme najít to jedno opravdu červené, a abychom dostali 10 bodů, zbývají nám poslední čtyři otázky. Teď už naše strategie nebude úplně symetrická a Vaškovy odpovědi nám budou vyrábět vždy dvě různé situace:

V jedenácté otázce se Vaška zeptáme na jablko z první hromádky a tři jablka z druhé. Pokud Vašek odpoví, že červené je mezi nimi, zahodíme zbylá jablka a dostaneme se do situace $(1, 3)$. Pokud odpoví, že jsou všechna zelená, dostaneme se do situace $(0, 8)$: Na druhé hromádce zůstalo 7 jablek, na něž jsme se neptali, a přibylo tam jedno jablko, které doteď bylo na první hromádce.

Pokud jsme v situaci $(0, 8)$, víme, že Vašek již lhal – žádné jablko není konzistentní se všemi jeho odpověďmi. Proto můžeme použít standardní binární vyhledávání a na tři otázky zjistit, které z našich $2^3 = 8$ jablek je to červené.

Zbývá tedy situace $(1, 3)$ a tři otázky na její vyřešení. Jedna možnost, jak strategii dokončit, je zeptat se Vaška na jediné jablko, to z první hromádky. Kladná odpověď znamená, že máme naše červené jablko (kdyby lhal, tak červené musí být nějaké z druhé hromádky, ale tam jsou jen jablka, o nichž už tvrdil, že jsou zelená, tj. kdyby byla skutečně červená, tak lhal již někdy dříve a lhát může jen jednou). Záporná odpověď nás dostane do situace $(0, 4)$, kterou pomocí binárního vyhledávání na dvě otázky vyřešíme.

Jiný pohled na vzorové řešení

Očíslujme si jablka čísly od 0 do 1023. Postupně položíme 10 otázek, v i-té z nich se zeptáme na těch 512 jablek, jejichž čísla mají i-tý bit roven 1. (Například první otázka se tedy ptá na jablka $1, 3, 5, 7, 9, \ldots$, druhá otázka se ptá na jablka $2, 3, 6, 7, 10, 11, \ldots$) Vždy, když Vašek odpoví ANO, zapíšeme si bit 1, když odpoví NE, zapíšeme si 0.

Takto dostaneme binární zápis čísla jediného jablka, které odpovídá všem 10 odpovědím. Buď je to opravdu červené jablko (a tedy Vašek ještě nelhal), anebo je právě jeden z těchto bitů špatně. Stejně jako v předchozím řešení jsme se tedy dostali do situace $(1, 10)$.

P-II-3 Turbojezdec

Podobně jako v domácím kole budeme úlohu řešit pomocí dynamického programování. Postupně pro každý počet tahů i a každou souřadnici (j, k) si spočítáme hodnotu $P[i, j, k]$ značící počet způsobů, kterými se můžeme dostat na políčko (j, k) tak, abychom navštívili ve správném pořadí prvních i písmen slova w.

Některé tyto hodnoty můžeme určit jednoduše. Označme jako $S[j, k]$ písmenko na políčku (j, k) na šachovnici. Pokud $S[j, k] \neq w[i]$, tak jistě $P[i, j, k] = 0$. Také pokud $S[j, k] = w[1]$, tak $P[1, j, k] = 1$.

Jak spočítat zbylé hodnoty $P[i, j, k]$? Skončit po i tazích na políčku (j, k) můžeme právě tak, že uděláme $i - 1$ tahů, těmi se dostaneme na nějaké jiné políčko a z toho následně i-tým tahem přejdeme na políčko (j, k). Způsoby, které nás tam dovedou z různých políček, jsou zřejmě navzájem různé, celkový počet způsobů tedy dostaneme tak, že sečteme počty způsobů pro všechny možnosti posledního tahu.

Formálně $P[i, j, k]$ spočítáme jakou součet všech $P[i - 1, j', k']$ takových, že turbojezdec umí jedním tahem přejít z (j', k') na (j, k). Řečením úlohy je potom součet $P[n, j, k]$ přes všechna políčka (j, k), kde n je délka slova w.

Na velké šachovnici umí turbojezdec skočit skoro odkudkoliv kamkoliv. Speciálně, pokud se políčka liší v každé souřadnici alespoň o 3, turbojezdec mezi nimi umí skočit, na každé políčko proto umí skočit z alespoň $(r - 5)(s - 5)$ jiných. To znamená, že přímý výpočet hodnoty $P[i, j, k]$ jako součet $P[i - 1, j', k']$ bude mít složitost alespoň $\mathcal{O}(rs)$, z čehož dostáváme celkovou složitost $\mathcal{O}(nr^2s^2)$.

Lepší řešení

Výše popsané řešení můžeme zlepšit tak, že budeme šikovněji sčítat možnosti, jak se na políčko dostat. Časovou složitost tím zlepšíme až na $\mathcal{O}(nrs)$, což stačí na plný počet bodů.

Naším hlavním nástrojem budou dvojrozměrné prefixové součty. Tato technika umí tabulku $r \times s$ čísel v čase $\mathcal{O}(rs)$ předzpracovat a potom v konstantním čase zjistit součet libovolného obdélníka. Pokud dvojrozměrné prefixové součty neznáte, detaily najdete v kuchařce KSP Základní algoritmy (*https://ksp.mff.cuni.cz/kucharky/zakladni-algoritmy/*).

Vždy, když spočítáme všechny hodnoty $P[i,j,k]$ pro konkrétní i a všechna j,k, vezmeme celou tabulku těchto hodnot, předpočítáme pro ni dvojrozměrné prefixové součty a pomocí nich budeme následně efektivněji počítat hodnoty $P[i+1,j,k]$.

Podívejme se nyní, odkud se turbojezdec umí dostat na konkrétní políčko (označené `X`). Na schématu níže jsou všechna taková políčka označená písmeny:

```
aaaaaaaab...deeee
aaaaaaaab...deeee
aaaaaaaab...deeee
cccccccc.....ffff
.................
..........X......
.................
gggggggg·····jjjj
hhhhhhhhi...kllll
```

Obecně umíme oblasti, odkud se turbojezdec umí dostat na políčko `X`, rozdělit na nejvýše 12 obdélníků (na schematu je každý obdélník označený jiným písmenem). Pokud v konstantním čase umíme zjistit součet libovolného obdélníka, umíme pak v konstantním čase zjistit i celkový počet způsobů, jak se po dalším tahu může turbojezdec dostat na dané políčko `X`.

Jiná možnost, jak se na tento součet podívat, je ta, že jde o (nejvýše) čtyři obdélníky, ale od každého musíme ještě odečíst to jeho rohové políčko, které je nejblíže políčku `X`.

```
def uvnitr(R, C, r, c):
    return 0 <= r < R and 0 <= c < C

def soucet(PS, r1, c1, r2, c2):
    r2, c2 = r2+1, c2+1
    return PS[r2][c2] - PS[r2][c1] - PS[r1][c2] + PS[r1][c1]

R, C = [ int(_) for _ in input().split() ]
W = input()
board = [ input() for r in range(R) ]

# Zjistíme počet způsobů (0 nebo 1) pro první písmeno
P = [ [ int( board[r][c] == W[0] ) for c in range(C) ] for r in range(R) ]

# Postupně pro každé další spočítáme prefixové součty
# a pomocí nich zjistíme nové počty
for n in range(1,len(W)):
    PS = [ [ 0 for c in range(C+1) ] for r in range(R+1) ]
    for r in range(R):
        for c in range(C):
            PS[r+1][c+1] = PS[r+1][c] + PS[r][c+1] - PS[r][c] + P[r][c]

    newP = [ [ 0 for c in range(C) ] for r in range(R) ]
    for r in range(R):
        for c in range(C):
            if W[n] != board[r][c]: continue
            if uvnitr(R, C, r-2, c-2):
                newP[r][c] += soucet(PS, 0, 0, r-2, c-2) - P[r-2][c-2]
            if uvnitr(R, C, r-2, c+2):
```

```
            newP[r][c] += soucet(PS, 0, c+2, r-2, C-1) - P[r-2][c+2]
        if uvnitr(R, C, r+2, c-2):
            newP[r][c] += soucet(PS, r+2, 0, R-1, c-2) - P[r+2][c-2]
        if uvnitr(R, C, r+2, c+2):
            newP[r][c] += soucet(PS, r+2, c+2, R-1, C-1) - P[r+2][c+2]

    P = newP

print( sum( sum(row) for row in P ) )
```

P-II-4 Vozidlo kóduje čísla

Podúloha a) – nalezněte předpis funkce

V našem pořadí máme všechny dvojice (x, y) uspořádané podle jejich součtu. Dvojice se stejným součtem jsou následně uspořádány podle x.

Pořadové číslo konkrétní dvojice je rovno počtu jiných dvojic, které jsou v pořadí před ní.

Před dvojicí (x, y) jsou všechny dvojice, jejichž součet je menší než $x+y$: jedna dvojice se součtem 0, dvě se součtem 1, ..., až $x + y$ dvojic se součtem $x + y - 1$. Těchto dvojic je tedy celkem

$$\Big(1 + 2 + \cdots + (x + y)\Big) = \frac{(x + y)(x + y + 1)}{2}.$$

Před dvojicí (x, y) jsou pak ještě dvojice se stejným součtem, ale menším prvním prvkem. Těch je x (první prvek v nich nabývá hodnot od 0 do $x-1$). Dohromady tedy platí:

$$spoj(x, y) = \frac{(x + y)(x + y + 1)}{2} + x.$$

Podúloha b) – naprogramujte funkci spoj

Makro snadno sestavíme pomocí existujících příkazů: do pomocných lokalit si uložíme hodnoty $x + y$ a $x + y + 1$, ty vynásobíme, výsledek vydělíme dvěma a přidáme x.

```
MAKRO spoj X Y Z
    zapiš X [X_plus_Y]
    přidej Y [X_plus_Y]
    zapiš [X_plus_Y] [X_plus_Y_plus_1]
    přidej J [X_plus_Y_plus_1]
    vynuluj [tmp]
    vynásob [X_plus_Y] [X_plus_Y_plus_1] [tmp]
    vynuluj [dva]
    přidej J [dva]
    přidej J [dva]
    vyděl [tmp] [dva] Z [zbytek]
    přidej X Z
KONEC
```

Podúloha c) – naprogramujte funkci první

Kdybychom měli nalézt matematický předpis této funkce, asi bychom s tím měli dost velké problémy a zřejmě by v takovém předpisu figurovala nějaká odmocnina a nějaké celé části. To by se pomocí našeho vozidla a kamenů počítalo hodně obtížně.

Naštěstí nic takového nepotřebujeme. Klíčem k úspěchu je nehledat *efektivní* řešení, ale *funkční a co nejjednodušší* řešení.

Začneme jednoduchým pozorováním. Před dvojicí (x, y) jsou určitě všechny dvojice $(0, 0)$, $(1, 0)$, $\ldots$, $(x - 1, 0)$, kterých je celkem x. Proto $spoj(x, y) \geq x$. Analogicky platí také $spoj(x, y) \geq y$.

Pokud dostaneme hodnotu $z = spoj(x, y)$ a hledáme hodnoty x, y, víme, že $x \leq z$ a zároveň $y \leq z$. To ale znamená, že máme jenom konečný počet možností a můžeme je tedy všechny projít a vyzkoušet.

V pseudokódu bude implementace funkce vypadat následovně:

Funkce *první*(z)

1. Pro všechna x od 0 do z:
2. Pro všechna y od 0 do z:
3. Jestliže $spoj(x, y) = z$:
4. Vrať hodnotu x.

Tento program nyní potřebujeme zapsat v jazyku, kterému rozumí naše vozidlo. Budeme tedy potřebovat dva do sebe vnořené cykly.

```
MAKRO první Z X
            vynuluj [px]
    cyklus1: vynuluj [py]

    cyklus2: spoj [px] [py] [zkouším]
             stejné [zkouším] Z odpověď
             skoč další2
    odpověď: zapiš [px] X

     další2: stejné Z [py] konec2
             přidej J [py]
             skoč cyklus2

     konec2: stejné Z [px] konec1
             přidej J [px]
             skoč cyklus1

     konec1: čekej
KONEC
```

Ve vnitřním cyklu si vždy pro aktuální hodnoty x a y do pomocné proměnné spočítáme $spoj(x, y)$ a porovnáme ji s hodnotou z. Když najdeme správnou dvojici x a y (vždy existuje právě jedna), zapíšeme x do výstupní proměnné. Cyklus ale pro jednoduchost nepřerušíme, necháme ho doběhnout.

Pokaždé, když hodnota y dosáhne z, vnitřní cyklus ukončíme a vrátíme se do vnějšího cyklu. Tam zvýšíme x, vynulujeme y a začneme novou iteraci vnitřního cyklu, nebo případně skončíme, jestliže jsme právě ukončili zkoušení možnosti $x = z$.

Funkce `druhý` zmíněná v podúloze d) by se od funkce `první` lišila pouze v jednom řádku: `odpověď: zapiš [py] Y`.

Podúloha d) – délka posloupnosti

Mějme číslo z, které představuje kód posloupnosti. Pokud je tímto číslem nula, jedná se o kód prázdné posloupnosti a ta má délku 0.

Je-li číslo z kladné, jistě platí $z = 1 + \mathit{spoj}(x_0, k)$, kde x_0 je první člen naší posloupnosti a k je kód jejího zbytku.

To ale znamená, že $z - 1 = \mathit{spoj}(x_0, k)$, jinými slovy, že $x_0 = \mathit{první}(z - 1)$ a $k = \mathit{druhý}(z-1)$. Pomocí již implementovaných funkcí tedy umíme z kódu neprázdné posloupnosti získat její první člen a také kód jejího zbytku.

Jaká je délka naší posloupnosti s kódem z? O jedna více než délka posloupnosti s kódem k.

Tuto úvahu můžeme v cyklu zopakovat: postupně po jednom budeme odebírat členy posloupnosti a zároveň si ve výstupní proměnné `X` budeme počítat, kolik jsme jich už odebrali. Skončíme, když nám zůstane prázdná posloupnost.

```
MAKRO délka Z X
            vynuluj X
            zapiš Z [posloupnost]
    cyklus: nulové [posloupnost] konec
            přidej J X
            # vypočítáme si z-1:
            přenes [posloupnost] J K -
            # z něho vypočítáme druhý(z-1):
            druhý [posloupnost] [zbytek_posloupnosti]
            # to je kód posloupnosti, která nám zůstala:
            zapiš [zbytek_posloupnosti] [posloupnost]
            skoč cyklus
     konec: čekej
KONEC
```