PRASK
VI. ročník, 2019/20
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola letnej časti

Mišof

1. Poskakujúci Super Mário

(max. 0 b za popis, 15 b za program)

Prvé dve podúlohy sú vlastne návodom k riešenie tretej, hlavnej podúlohy. Prejdeme si preto detailne cez prvú podúlohu a potom rovno ukážeme všeobecné riešenie tretej. Vo všetkých riešeniach budeme dva smery pohybu pre jednoduchosť volať "doprava" a "doľava", netreba však zabúdať, že stanica je cyklická – teda keď Mário pôjde dostatočne dlho "doprava", dostane sa určite na miesto, kde začínal.

Podúloha A: je miestnosť jedna?

Jediná vec, ktorú vie Mário použiť na rozlíšenie miestností, je svetlo. Ako ale spoznať, či je to naozaj nová miestnosť a nie iná, v ktorej je tiež zasvietené?

Trik spočíva v tom, že ak ide o dve rôzne miestnosti, prepnutie svetla v jednej z nich sa neprejaví v druhej. Keď si toto uvedomíme, už ľahko vymyslíme spôsob kontroly. Jedno možné riešenie vyzerá nasledovne:

```
1. rozsvieť
2. pohni sa doprava
3. ak tam je zhasnuté, je to iná miestnosť, a teda sú aspoň dve, koniec
4. ak tam je rozsvietené, zhasni
5. pohni sa naspäť doľava
6. ak je tam zrazu zhasnuté, muselo ísť o tú istú miestnosť, a teda je len jedna, koniec
7. ak v prvej miestnosti ostalo rozsvietené, sú aspoň dve miestnosti, koniec
```

Podúlohy B a C: koľko je miestností?

Jedno možné všeobecné riešenie podúlohy C bude vyzerať tak, že postupne otestujeme, či sú miestnosti aspoň dve (to už vieme robiť), či sú aspoň tri, aspoň štyri, a tak ďalej. Akonáhle takýto test prvýkrát zlyhá, budeme presne vedieť, koľko miestností má náš systém.

Teraz si rozmyslíme, ako skontrolovať, či je miestností aspoň k. Postup bude zovšeobecnením toho z podúlohy A: najskôr dostatočne veľa miestnosti v rade rozsvietime a potom sa pozrieme, čo sa stane, keď ich pôjdeme postupne zhášať. Ukážeme si najskôr samotné riešenie:

```
1. rozsvieť v miestnosti, kde začínaš
2. (k-1)-krát sa posuň doprava a rozsvieť aj tam (ak bolo zhasnuté)
3. zhasni v miestnosti, kde práve si
4. (k-1)-krát sprav nasledovné:
       posuň sa naspäť doľava
       ak si prišiel/a do miestnosti, kde je zhasnuté, miestností je menej ako k, koniec
       zhasni v miestnosti, kde práve si
5. ak si sa dostal/a až sem, miestností bolo aspoň k
```

Prečo toto riešenie funguje? Rozmyslime si, čo sa stane, ak máme miestností aspoň k a čo sa stane, ak ich je menej. Ak máme miestností aspoň k, v prvých dvoch krokoch postupne navštívime k rôznych miestností a v každej z nich rozsvietime, a potom v krokoch 3 a 4 cez ne prejdeme naspäť a v každej zhasneme. Všetko zafunguje ako má a dáme správnu odpoveď.

Ak máme miestností menej ako k, niekedy počas kroku 2 sa "zacyklíme" – teda prídeme znova do miestnosti, v ktorej sme už boli. Špeciálne teda aj miestnosť, v ktorej skončíme krok 2, sme určite navštívili (aspoň) dvakrát. V kroku 3 v nej zhasneme. Ale potom keď sa v kroku 4 vraciame naspäť, určite sa ešte (aspoň) raz do tejto miestnosti vrátime. No a keď sa tak stane, uvidíme, že v nej už je zhasnuté a z toho vieme, že sme v nej už boli, a teda že miestností musí byť menej ako k.

Podúloha D: efektívne riešenie

Koľko zhruba krokov urobí Mário pri predchádzajúcom riešení? Ak je skutočný počet miestností n, Mário postupne spraví 1 krok doprava, 1 doľava, 2 doprava, 2 doľava, ..., až nakoniec n doprava a potom n doľava. Dokopy teda spraví rádovo n^2 krokov (a rádovo toľko isto prepnutí svetla).

Ako vieme tento postup urýchliť? Všimnime si, čo sa stane, ak použijeme ten istý postup, len nebudeme skúšanú veľkosť miestnosti zväčšovať vždy o 1, ale rýchlejšie. Napríklad sa zamyslime, čo sa stane, keď vyššie popísaný postup použijeme pre $k = 10$, ale miestností je len 7. V krokoch 1 a 2 postupne navštívime 10 miestností a v každej zažneme. V krokoch 3 a 4 sa nám potom ale podarí zhasnúť len 7-krát (lebo len toľko miestností naozaj máme) a po nasledujúcom pohybe už nájdeme v miestnosti tmu.

Ak teda použijeme vyššie použitý postup a budeme si navyše počítať, v koľkých miestnostiach sa nám naozaj podarilo zhasnúť, vieme presne povedať, či bola miestnosť len jedna, či boli presne dve, presne tri, ..., či ich bolo presne $k - 1$ alebo či ich muselo byť aspoň k. No a po tomto vylepšení už môžeme skúšať len niektoré vhodne zvolené hodnoty k.

Celé riešenie teda bude vyzerať napríklad tak, že vyššie popísaným postupom budeme postupne zisťovať presný počet miestností ak je menší ako 2, ak je menší ako 4, potom 8, 16, 32, a tak ďalej. (Rovnako dobre sme mohli použiť napríklad mocniny 10, alebo inú postupnosť ktorá rastie exponenciálne.)

Prečo má tento postup lepšiu časovú zložitosť? Počítajme. V poslednom kole (v tom, kedy už odhalíme správny počet miestností) prejdeme celý systém nanajvýš dvakrát smerom tam a presne raz smerom späť. V predposlednom kole prejdeme nanajvýš celý systém tam a späť. O kolo skôr to už muselo byť nanajvýš pol systému tam a späť, predtým štvrtina, a tak ďalej. No a keď toto celé spočítame, dostaneme, že dokopy spravíme nanajvýš $7n$ krokov. A keďže každému kroku zodpovedá najviac jedno prepnutie svetla, je aj tých len lineárne veľa – a teda celé toto riešenie má časovú zložitosť lineárnu od počtu miestností v stanici.

Formálne, k ľubovoľnému n vieme nájsť t také, že $2^{t-1} \leq n < 2^t$. Pre toto n spraví vyššie popísaný algoritmus postupne t kontrol, pričom bude postupne voliť $k = 2^1, 2^2, \ldots, 2^t$. Počas týchto kontrol spraví menej ako $2^1 + 2^2 + \cdots + 2^t < 2^{t+1}$ krokov smerom doprava a menej ako $2^1 + 2^2 + \cdots + 2^{t-1} + n < 2^t + n$ krokov smerom doľava. Dokopy teda spraví menej ako $2^{t+1} + 2^t + n = 6 \cdot 2^{t-1} + n \leq 6n + n = 7n$ krokov.

Paulinka

2. Rebríky a Platformy

(max. 0 b za popis, 15 b za program)

V prvých dvoch podúlohách ste si mohli vyskúšať ako riešiť úlohu manuálne, a potom toto riešenie zovšeobecniť pre ľubovoľnú zadanú postupnosť stúpaní a klesaní.

Podúloha (a)

- Príklad takejto postupnosti môže byť napríklad 4, 3, 1, 5, 2, 6. Všimnite si, že rovnako dobré by bolo riešenie 4, 3, 1, 6, 2, 5, či 5, 3, 1, 4, 2, 6.
- Všimnite si, že tu musí byť stredná platforma vo výške 1, pretože všetky ostatné platformy sú od nej nižšie. Pre ostatné už máme viac možností, napríklad 3, 2, 1, 4, 5, či 5, 4, 1, 2, 3.
- Tu riešením môže byť napríklad 3,2,1,4,5,6,12,11,10,7,8,9

Podúloha (b)

Prvé pozorovanie je, že jednotkovú výšku môžu mať iba platformy z ktorých idú oba (prípadne len jeden, ak to je koncová platforma) rebríky hore.

Platí však, že každá takáto platforma môže mať výšku jedna?

Predstavme si že nastavíme výšku tejto platformy na jedna. Teraz sa nám dráha rozdelí na dve časti (časť môže byť aj prázdna). Všimnime si, že nech sú vedľajšie platformy akejkoľvek zostávajúcej výšky (teda výsky od dva do počtu platforiem), tak nerovnosť s najnižšou platformou stále platí!

Teraz si vieme povedať, že do jednej časi dráhy (napríklad tej naľavo od najnižšej platformy) dáme výšky od 2 do $2+n$ (kde n je počet platforiem naľavo od nás) a do pravej tie zvyšné. Takto sa nám problém rozpadne na dva podproblémy, a ak vždy existuje riešenie (ako ukážeme, existuje), jednotka na danom mieste môže byť.

Podúloha (c)

Predchádzajúca podúloha nám povedala ako na to: nájdime ľubovoľnú platformu ktorá môže mať výšku jedna. Potom sa pozrime na platformy naľavo od nej. Ak tam už nie sú žiadne, zopakujeme nasledovný proces pre pravú stranu:

Znova zopakujeme postup: nájdeme platformu ktorá môže mať výšku dva. To je ľubovoľná z ktorej idú všetky (jeden alebo dva) rebríky hore, pričom sa tvárime že rebrík na platformu jedna neexistuje (vieme, že nech bude výška susediacej platformy ľubovoľná z 2 až po počet platforiem, nerovnosť je vždy splnená). Teraz zase rozdelíme stranu na dve časti a ak je niektorá z nich neprázdna opakujeme tento postup.

Keď sme skončili ľavú stranu, opakujme pre pravú stranu.

Pre ilustráciu sa pozrime na podúlohu (a), tretiu dráhu:

Tretia platforma môže byť najnižšie, takže nech má výšku jedna. Potom máme ↓↓ na ľavej strane, takže platforma s výškou dva musí z týchto dvoch byť tá druhá a prvá platforma má tak výšku 3.

Dokončili sme ľavú stranu, pozrime sa na pravú. Hneď krajná (štvrtá platforma) môže mať výšku 4. Potom piata má zas len rebrík hore, takže môže mať výšku 5 a podobne šiesta môže mať výšku 6.

Zo siedmej už ide rebrík dole, tak musíme hľadať ďalej. Jediné miesto, kde vieme umiestniť výšku 7 je teraz desiata platforma. Platformy naľavo od nej postupne (v poradí siedma, ôsma, deviata) dostanú výšky 10, 9 a 8.

A napokon, posledné dve platformy skončia s výškami 11 a 12, a teda výsledné výšky sú 3,2,1,4,5,6,10,9,8,7,11,12.

Podúloha (d)

Ako vysoko môže byť prvá platforma? Môže byť vo výške 1? Ak z nej nejde rebrík dole, potom áno (ako sme ukázali v podúlohe (b)). Čo ak z nej ide dole za sebou k rebríkov? Potom môže byť najnižšie vo výške $k+1$ – nasledujúce platformy musia byť postupne vo výškach k, $k-1$, atď. až po 1. Potom si zvyšok platformy vieme predstaviť ako separátnu dráhu, do ktorej umiestňujeme výšky $k+1$ až po počet platforiem.

Podobne sa môžeme pýtať, či môže byť prvá platforma vo výške n. Tiež platí, že ak prvých k rebríkov ide hore, môže byť prvá platforma vo výške nanajvýš $n-k$.

Čo s ostatnými výškami?

Bez ujmy na všeobecnosti, nech ako prvé ide k rebríkov dole. Dajme prvú platformu do ľubovoľnej výšky aspoň $k+1$ a ďalšie k platformy budú postupne k, $k-1$, atď až po 1. Potom akúkoľvek výšku bude mať nasledovná platforma, tak nerovnosť bude platiť (keďže sme výšku jedna umiestnili tak kde môže byť). Potom nám vlastne vznikol separátny problém s $n-k-1$ platformami, ktorý určite vieme algoritmom z úlohy (c) vyriešiť.

Takže akákoľvek výška $> k$ pre prvú platformu funguje. Rovnako, keď z prvej platformy ide k rebríkov hore, na platforme jedna môže byť akákoľvek výška $\leq n-k$.

Podúloha (e)

Z podúlohy (d) vieme, akú najvyššiu výšku vieme dať na prvú platformu.

Najskôr sa pozrime na prípad, že prvý rebrík ide dole. Vtedy je najvyššia platforma hneď na začiatku. Potom keď sa pozrieme na zvyšné platformy, dostaneme rovnaký problém ako predtým, len o jednu platformu kratší.

Ak ide z prvej platformy k rebríkov po sebe hore, potom najvyššie ako vie byť prvá platforma je $n-k$, a nasledujúce platformy musia byť $n-k+1$, $n-k+2$, …, n. Ďalšia platforma musí byť nižšia – dostali sme menší problém s $n-k-1$ platformami.

Ako tento algoritmus funguje si ukážme znova na poslednom príklade z podúlohy (a):

Máme dvanásť platforiem. Prvé dva rebríky idú dole, takže najvyššie ako vieme položiť prvé dve platformy sú postupne 12 a 11 metrov. Ďalšie štyri rebríky idú hore, takže ďalších päť platforiem je vo výškach postupne 6, 7, 8, 9 a 10 metrov. Ďalšie tri rebríky idú dole, takže najväčšie možné výšky ďalších troch platforiem sú 5, 4 a 3. A napokon, posledné dva rebríky smerujú hore, takže posledné dva rebríky musia byť 1 a 2.

Emo

3. Nuda na štrkovisku

(max. 0 b za popis, 15 b za program)

Na vstupe sme dostali už utriedenú postupnosť, takže jediné čo potrebujeme spraviť je odsimulovať hru Jitky a Marcela. Jeden z problémov, na ktorý sme mohli naraziť je že súčet hmotností kamienkov je veľmi veľký. Tento súčet sa nezmestí do bežnej premennej v C++ alebo Pascale. V tomto príklade mali výhodu riešitelia, ktorí programovali v Pythone, pretože v Pythone majú celočíselné premenné ľubovolný rozsah. V ostatných jazykoch stačilo však použiť 64 bitové premenné (do ktorých sa tento súčet zmestí) a všetko bolo v poriadku.

Časová aj pamäťová zložitosť tohto algoritmu je lineárna v závislosti od n, čo zapisujeme ako $O(n)$, pretože iba v jednom cykle spočítame súčet hmotností kamienkov pre Jitku a Marcela a sčítavanie je konštantná operácia.

Listing programu (Python)

```
n = int(input())
k = map(int, input().split())

j, m = 0, 0
for x in k:
  if j <= m:
    j += x
  else:
    m += x

if j == m:
  print('Remiza')
elif j < m:
  print(f'Marcel {m - j}')
else:
  print(f'Jitka {j - m}')
```

Listing programu (C++)

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

int main() {
  int n; cin >> n;

  vector<int> k(n);
  for (int i = 0; i < n; i++) cin >> k[i];

  long long j = 0, m = 0;
  for (int i = 0; i < n; i++) {
    if (j <= m) j += k[i];
    else m += k[i];
  }

  if (j == m) cout << "Remiza";
  else if (j > m) cout << "Jitka " << j - m;
  else cout << "Marcel " << m - j;
  cout << endl;
}
```

Iné riešenie

Problému s veľkými číslami sa dalo vyhnúť ak sme si v programe počítali iba rozdiel skóre. Stačí si uvedomiť, že v ľubovolnom momente počas behu programu tento rozdiel nemôže byť v absolútnej hodnote väčší ako najťažší kamienok (lebo inak by musel vyberať kamienok hráč s väčším skóre).

Listing programu (C++)

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

int main() {
  int n; cin >> n;

  vector<int> k(n);
  for (int i = 0; i < n; i++) cin >> k[i];
```

```
  int diff = 0;
  for (int i = 0; i < n; i++) {
    if (diff <= 0) diff += k[i];
    else diff -= k[i];
  }

  if (diff == 0) cout << "Remiza";
  else if (diff > 0) cout << "Jitka " << diff;
  else cout << "Marcel " << -diff;
  cout << endl;
}
```

4. Sekanie bagety

15 b za popis, 0 b za program

Pomalé riešenie

Priamočiary spôsob ako nájsť najchutnejší úsek je pre každý možný úsek zrátať jeho chuť a z nich potom vybrať ten najlepší. Označme si teda u_i chuť úseku, ktorý začína na i-tej pozícii. Tieto hodnoty potom vieme vypočítať ako:

$$u_1 = c_1 + c_2 + \cdots + c_k$$

$$u_2 = c_2 + c_3 + \cdots + c_{k+1}$$

$$\cdots$$

$$u_{n-k+1} = c_{n-k+1} + c_{n-k+2} + \cdots + c_n$$

Následne nám stačí vybrať to najväčšie.

Pre každý úsek musíme sčítať k čísel a úsekov je dokopy $n-k+1$. To nám dáva časovú zložitosť $O(k(n-k))$, čo je v najhoršom prípade $O(n^2)$.

Listing programu (C++)

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

int main()
{
        int n, k;
    cin >> n >> k;
    vector<int> v(n);
    for(int i = 0; i < n; i++) cin >> v[i];
    long long maxSucet = -9999999999999999LL;
    for(int i = 0; i < n-k+1;i++)
    {
        long long sucet = 0;
        for(int j = 0; j < k; j++) sucet+=v[i+j];
        maxSucet = max(sucet,maxSucet);
    }
    cout << maxSucet << "\n";
}
```

Vzorové riešenie

Aby sme dosiahli rýchlejšie riešenie, môžeme sa pokúsiť zlepšiť spôsob akým počítame chuť jednotlivých úsekov. Na to si stačí uvedomiť, že dva po sebe idúce úseky spolu zdieľajú väčšinu dielikov:

$$u_1 = c_1 + (c_2 + \cdots + c_k)$$

$$u_2 = (c_2 + \cdots + c_k) + c_{k+1}$$

Z toho vidno, že:
$u_2 = u_1 - c_1 + c_{k+1}$
Takto môžeme pokračovať:

$$u_3 = u_2 - c_2 + c_{k+2}$$

$$\cdots$$

$$u_{n-k+1} = u_{n-k} - c_{n-k} + c_n$$

Hodnotu u_1 musíme stále vypočítať normálne (sčítaním k čísel) v čase $O(k)$. Zvyšných $n - k$ úsekov už ale vieme každý spočítať v konštantnom čase, výsledkom čoho bude časová zložitoť $O(n)$.

Listing programu (C++)

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

int main()
{
        int n, k;
    cin >> n >> k;
    queue<long long> Q;
    long long sucet = 0;
    long long maxSucet = -9999999999999999LL;
    for(int i = 0; i < n; i++)
    {
        long long x;
        cin >> x;
        sucet+=x;
        Q.push(x);
        if(Q.size() > k)
        {
            sucet -= Q.front();
            Q.pop();
        }
        if(Q.size() == k) maxSucet = max(sucet, maxSucet);
    }
    cout << maxSucet << "\n";
}
```

Listing programu (Python)

```
n, k = list(map(int, input().split()))
cisla = list(map(int, input().split()))

maxSucet = -9999999999999999
sucet = 0
for i in range(n):
    sucet += cisla[i]
    if i >= k:
        sucet -= cisla[i-k]
    if i >= k-1:
        maxSucet = max(sucet, maxSucet)

print(maxSucet)
```

5. Krutý súboj planét

15 b za popis, 0 b za program

Bez žolícia

Keďže nie sú žiadne bonusové suroviny, robotov treba vyrobiť iba z normálnych surovín. Z každej suroviny treba použiť na výrobu jedného robota určený počet, takže stačí zisťovať koľko najviac sa dá poskladať z každej suroviny robotov. Keď je niektorej suroviny dostatok len na k robotov, tak určite sa nedá postaviť viac ako k robotov, lebo z danej suroviny nie sú suroviny ani na robota naviac. Teda celkovo najväčší počet robotov je minimum z počtov robotov, ktorých sa dá poskladať z nejakej suroviny. Ak ide o obranu, je potrebné iba zistiť, či je to dostatočný počet alebo nie. Ak ide o útok, výstupom je spomínaný počet. Spracovanie každej suroviny zaberie len niekoľko numerických operácií, teda časová zložitosť je $O(n)$.

Len obrana

Keďže je presne určené koľko robotov treba vyrobiť, je známe aj koľko na to treba z každej suroviny. Jednoducho sa teda dá zistiť, koľko z každej suroviny chýba a teda je potrebné doplniť žolíciom. Ak by chýbalo viac ako b surovín, nedá sa vyrobiť daný počet robotov, keďže nie je dostatok žolícia. Inak je možné vyrobiť toľko robotov. Toto sa dá opäť zistiť v $O(n)$, keďže je potrebné niekoľko numerických operácií na spracovanie každej suroviny.

Pomalé riešenie

Pomocou algoritmu na obranu sa teraz dá postupne skúšať či je možné postaviť jedného robota, dvoch, … Takto sa určite raz stane, že nebude možné postaviť daný počet robotov, lebo nebude dostatok nejakej suroviny na postavenie daného počtu robotov, teda počet robotov už nebude väčší, čím je zistený maximálny počet robotov. Problémom ale je, že počet robotov môže byť veľmi veľký a teda riešenie bude pomalé.

Vzorové riešenie

K vzorému riešeniu chýba už len rozvinúť vyššie načrtnutú myšlienku – postupným skúšaním či sa dá daný počet robotov vyrobiť. Najprv bude odpoveď áno, kým je možné daný počet vyrobiť. Potom nastane zmena a odpoveď bude vždy nie. Vďaka tomu nie je potrebné skúšať všetky možné počty robotov, ale dá sa binárne vyhľadať najväčší počet robotov, ktorý je možné vyrobiť. Stačí si pamätať aký najväčší a aký najmenší počet robotov je možné vyrobiť. Na začiatku je určite možné vyrobiť 0 robotov, a zároveň určite nie je možné vyrobiť viac ako x robotov, kde x je najväčší počet robotov, napríklad súčet všetkých surovín plus jedna.

Vždy bude platiť, že minimum označuje počet, ktorý sa dá vyrobiť a maximum počet, ktorý sa už vyrobiť nedá. Tieto dve hodnoty tvoria nejaký úsek, interval hodnôt, ktoré by mohli byť odpoveďou na našu otázku. Takýto interval však vieme po krokoch zlepšovať. V každom kroku je potrebné zistiť, či sa dá vyrobiť počet robotov, ktorý je v strede medzi krajnými hodnotami úseku. Ak sa daný počet vyrobiť dá, posunieme minimum koľko sa dá vyrobiť na túto hodnotu. Ak sa daný počet nedá vyrobiť, posunieme maximum. Po každom kroku bude úsek polovičný, to znamená, že dokopy spravíme určite najviac logaritmus krokov. To, prečo je to naozaj tak, je nad rámec tohto vzoráku. Pamäťová zložitosť je $O(n)$, časová $O(n \log x)$,

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool dasa(const vector<long long>& mam, const vector<long long>& treba, long
    ↪ long kolko, long long b) {
        for (int i = 0; i < mam.size(); ++i) {
                b -= max(0ll, treba[i]*kolko-mam[i]);
                if (b < 0) return false;
        }
        return true;
}

int main() {
```

```
        string s;
        long long n, b;
        cin >> s >> n >> b;

        vector <long long> mam(n), treba(n);

        for (int i = 0; i < n; ++i) cin >> mam[i];
        for (int i = 0; i < n; ++i) cin >> treba[i];

        if (s == "Obrana" ) {
                long long kolko;
                cin >> kolko;

                if (dasa(mam, treba, kolko, b)) cout << "Ano\n";
                else cout << "Nie\n";

        } else {
                long long l = 0, r = 10000000000000;
                while(r-l-1) {
                        long long m = (l+r)/2;
                        if (dasa(mam, treba, m, b)) l = m;
                        else r = m;
                }
                cout << l << '\n';
        }
        return 0;
}
```

Listing programu (Python)

```
s = input()
n, b = map(int,input().split())
mam = [int(i) for i in input().split()]
treba = [int(i) for i in input().split()]
maximum = 10000000000000

def dasa(kolko, b):
  global mam, treba
  for i in range(n):
    b -= max(0, treba[i]*kolko-mam[i])
    if b < 0: return False
  return True

if s == "Obrana" :
  kolko = int(input())

  if dasa(kolko, b):
    print("Ano")
  else:
    print("Nie")

else:
  l = 0
  r = maximum+1
  while r-l > 1:
    m = (l+r)//2
    if dasa(m, b):
      l = m
```

```
    else:
      r = m
  print(l)
```