Korešpondenčný seminár z programovania
XXXVII. ročník, 2019/20
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola zimnej časti

Sabinka a Marcel

1. Dvojičky

(max. 12 b za popis, 8 b za program)

Je niekoľko spôsobov, ako sa na túto úlohu pozerať. Líšia sa časovou a pamäťovou zložitosťou.

Priamočiare riešenie

Pre každý možný začiatok intervalu potrebujeme zistiť, aký je súčet prvkov po posledný a následne z nich vybrať maximálny. Toto riešenie zaberie $O(n)$ pamäte, lebo si pamätáme celé pole, a $O(n^2)$ času, lebo pre každý prvok musíme sčítať všetky ďalšie. Dokopy, za celý beh programu počet prvkov, ktoré potrebujeme sčítať vieme vyjadriť ako $\frac{n \cdot (n-1)}{2}$, čo je asymptoticky n^2.

Listing programu (Python)

```
from math import inf
n = int(input())
p = list(map(int, input().split()))

max_suma = -inf
for i in range(len(p)):
    suma = 0
    for j in range(i, len(p)):
        suma += p[j]
    max_suma = max(suma, max_suma)

print(max_suma)
```

Listing programu (C++)

```
#include<stdio.h>

int main()
{
    long long n=0, i=0, j=0, best=-1000000000000, sum=0;
    scanf("%lld", &n);
    long long pole[n];
    for(i=0;i<n;i++)
        scanf("%lld", &pole[i]);

    for(i=0;i<n;i++)
    {
        sum=0;
        for(j=i;j<n;j++)
            sum+=pole[j];
        if(sum>best)
            best=sum;
```

```
    }
    printf("%lld\n", best);
    return 0;
}
```

Ako zlepšiť časovou zložitosť

To, na čom náš kód trávi priveľa času, je to, že každý interval počíta odznovu. Na to, aby sme nemuseli vždy počítať súčet celého nového intervalu, si stačí uvedomiť, že ako prechádzame intervaly od najdlhšieho, tak nový interval je vždy len o jeden prvok kratší ako ten starý, resp., ten starý je o jeden prvok dlhší. To znamená, že ak sa budeme pozerať na intervaly od najkratšieho, (ako prvý sa pozrieme na ten, ktorý má len jeden prvok), tak súčty všetkých intervalov, ktoré končia na poslednom prvku poľa zistíme na n krokov, kde v každom kroku pripočítame k aktuálnemu súčtu ďalší prvok. Pamäťová zložitosť bude $O(n)$, lebo stále si musíme pamätať celé pole, ale časová zložitosť bude už iba $O(n)$.

Listing programu (Python)

```
from math import inf
n = int(input())
p = list(map(int, input().split()))

max_suma = -inf
suma = 0
for i in range(n-1, -1, -1):
    suma += p[i]
    max_suma = max(max_suma, suma)



print(max_suma)
```

Listing programu (C++)

```
#include<stdio.h>

int main()
{
    long long n=0, i=0, best=-1000000000000, sum=0;
    scanf("%lld", &n);
    long long pole[n];
    for(i=0;i<n;i++)
        scanf("%lld", &pole[i]);

    for(i=n-1;i>=0;i--)
    {
        sum+=pole[i];
        if(sum>best)
            best=sum;
    }
    printf("%lld\n", best);
    return 0;
}
```

Ako to robiť bez nutnosti pamätania všetkých súčtov

Toto riešenie samozrejme môže mať veľa variácií, napríklad namiesto hľadania intervalu s čo najväčším súčtom, ktorý končí na poslednom prvku poľa, môžeme hľadať interval od začiatku, ktorého suma je čo najmenšia. V takomto prípade je výsledok súčet celého poľa mínus súčet intervalu s najmenším súčtom.

Toto dokonca dokážeme urobiť v pamäti $O(1)$, a čase $O(n)$, ak načítavame prvky postupne, po jednom.

Listing programu (C++)

```
#include<stdio.h>


int main()
{
    long long n=0, i=0, best=0, sum=0, temp=0;
    scanf("%lld", &n);
    for(i=0;i<n;i++)
        {
                scanf("%lld", &temp);
                sum+=temp;
                if(i!=n-1)
                        if(sum<best)
                                best=sum;
    }
    printf("%lld\n", sum-best);
    return 0;
}
```

Kika

2. Odstrihnuté útvary

(max. 4 b za popis, 16 b za program)

Ako ste si mohli všimnúť, v tejto úlohe boli viditeľné vstupy a výstupy. Teda na správne riešenie za 16 bodov stačilo zistiť všetky priestorové a všetky rovinné útvary, ktoré boli na vstupe a následne napísať program, ktorý si ich všetky zapamätá.

Toto riešenie sa dá jednoducho skrátiť ak si všimneme pár zákonitostí. Až na pár výnimiek sa rovinné útvary končia príponou `„uholnik"` a priestorové sa končia `„sten"`. Programu už teda stačí zapamätať si len niekoľko výnimiek. V prvej sade neboli žiadne výnimky a teda na 4 body za prvú sadu stačilo rozoznať útvary končiace na `„uholnik"` a útvary končiace na `„sten"`.

Aby sme program ešte skrátili bude najlepšie ak sa pohráme len s písmenkami. Môžeme si všimnúť, že všetky útvary končiace na `„n"` okrem `„oktagon"` a `„hexagon"` sú priestorové. Žiadny rovinný útvar ale nemá piate písmeno `„g"`. Vieme teda použiť, že všetky končiace na `„n"`, ktoré nemajú piate písmeno `„g"` sú priestorové. Všetky útvary končiace na písmeno `„k"` sú rovinné. Teraz nám stačí sa pozrieť len na výnimky. Z výnimiek, `„gula"` a všetky slová začínajúce na `„k"`, okrem slova `„kruznica"` sú priestorové. Zvyšné sú rovinné.

Listing programu (C++)

```
#include<iostream>
#include<string>

using namespace std;

int main() {
   string x; cin >> x;
   int k = x.size() - 1;
   if (x[0] == 'g' || (x[0] == 'k' && x[1] != 'r') || (x[k] == 'n' && x[k-2] != 'g')) cout << "priestorovy" << endl;
   else cout << "rovinny" << endl;
}
```

Niektorí z vás našli ďalšie pravidlá, ktoré sa dajú využiť. Stačí sa napríklad pozrieť iba na tretie a piate písmenko odzadu.

Veľmi krátke riešenie, sa dalo získať aj pomocou regulárnych výrazov, ktorými vieme na pomerne málo znakov popísať text, ktorý potom možeme vo vstupnom reťazci vyhľadávať. Napríklad regulárny výraz `sten$|uholnik|..ica`

popisuje výraz, ktorý končí (symbol `$`) na `sten` alebo (symbol |) obsahuje `uholnik` alebo obsahuje text s ľubovolnými dvoma písmenami (symbol `.`) nasledujúc textom `ica`. *(Knižnice pre regulárne výrazy obsahuje každý slušný jazyk, vrátane Pythonu a C++11 a dokážu toho omnoho viacej. Podrobný popis regulárnych výrazov nájdete na internete).*

Listing programu (Python)

```
import re
print(re.search("k$|go|c.?$|^u",input()) and "rovinny" or "priestorovy")
```

Ďalšie nápady už necháme na vás.

Danza

3. Lacný polozisk

(max. 12 b za popis, 8 b za program)

Za úlohu sme mali prejsť z ľavého horného do pravého dolného políčka mriežky tak, aby sme maximalizovali súčet hodnôt prejdených políčok. Mohli sme pri tom chodiť iba hore, dole a doprava.

Jednoducho, neefektívne, funkčne

Ak nevieme, čo s úlohou, niet lepšieho nápadu, ako napísať riešenie hrubou silou. Možno si potom niečo všimneme a budeme takéto riešenie vedieť prerobiť na nejaké rýchlejšie.

Nuž, keďže chceme hrubú silu, vyskúšajme všetky možnosti. Ináč povedané, skúsme postupne každú možnú cestu z ľavého horného, na pravé dolné políčko. Samozrejme, chceme skúšať iba tie cesty, kde nechodíme doľava, pretože to predsa nesmieme. Pre každú cestu zrátame súčet hodnôt prejdených políčok. Z nich nakoniec vypíšeme maximum.

Ako niečo takého naprogramovať? Jednoduchým riešením je rekurzia. O každom políčku si budeme pamätať, či už sme cez neho prešli, alebo nie. Tiež si budeme pamätať, na ktorom políčku aktuálne stojíme.

Aké máme možnosti, keď stojíme na políčku $[r, c]$? Môžeme sa pohnúť hore, teda na políčko $[r-1, c]$, dole, teda na políčko $[r+1, c]$, alebo doprava, teda na políčko $[r, c+1]$. Samozrejme, ak je dané políčko už označené ako navštívené, alebo ak neexistuje (teda je mimo mriežky), tak naň stúpiť nemôžeme. Ktorú možnosť si vybrať? Keďže skúšame všetky možnosti, tak si postupne vyberieme všetky. Políčko, na ktoré sa rozhodneme ísť označíme ako navštívené. K aktuálnemu súčtu si pripočítame hodnotu daného políčka a rekurzívne sa zavoláme na to políčko. Ked sa vrátime z rekurzie, označíme ho opäť ako nenavštívené a skúsime ďalšiu možnosť. Takto postupne preveríme všetky 3 možnosti pre dané políčko.

Samozrejme, ak sa rekurziou zavoláme až na políčko $[R-1, C-1]$, tak sa ďalej nevoláme, ale iba si zapamätáme maximum z doterajšieho maxima a aktuálneho súčtu políčok a vrátime sa z rekurzie.

Technika, ktorú sme použili sa volá rekurzívny backtracking a na napísanie riešenia hrubou silou sa dá použiť veľmi často.

Akú to má zložitosť? Keďže skúšame všetky cesty, tak to určite bude aspoň toľko, koľko je ciest. A koľko to je? Nuž, to nie je celkom jednoduché.

Ľahko však spravíme horný odhad. Na každom z RC políčok skúšame 3 možnosti. Každú cestu aj naozaj prejdeme. Najdlhšia možná cesta má nanajvýš RC políčok. Horný odhad teda bude $O(RC \cdot 3^{RC})$.

Môžeme spraviť aj dolný odhad. Keby sme chodili iba dole a doprava, tak by sme dokopy vždy spravili $R+C$ krokov. Práve R z nich bude doprava. Počet takýchto ciest teda bude $(R+C) \cdot \binom{R+C}{R}$. Tým, že pridáme možnosť chodiť aj hore nám počet týchto ciest nezníži.

Časová zložitosť nášho programu je niekde medzi tým. Malo by to stačiť na prvú sadu.

Pamäť je $O(RC)$.

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

const ll MAXR = 1111, MAXC = 1111;
ll R, C;
ll G[MAXR][MAXC];
bool vis[MAXR][MAXC];

enum {LEFT, TOP, BOTTOM};

ll go(ll r, ll c) {
    if(r == R - 1 && c == C - 1)
        return 0;
    ll ans = INT_MIN;
```

```
    // Chod doprava, ak mozes
    if(c < C - 1) {
        vis[r][c + 1] = true;
        ans = max(ans, G[r][c + 1] + go(r, c + 1));
        vis[r][c + 1] = false;
    }
    // Chod dole, ak mozes
    if(r < R - 1 && !vis[r + 1][c]) {
        vis[r + 1][c] = true;
        ans = max(ans, G[r + 1][c] + go(r + 1, c));
        vis[r + 1][c] = false;
    }
    // Chod hore, ak mozes
    if(r > 0 && !vis[r - 1][c]) {
        vis[r - 1][c] = true;
        ans = max(ans, G[r - 1][c] + go(r - 1, c));
        vis[r - 1][c] = false;
    }
    return ans;
}

int main() {
    cin >> R >> C;
    for(ll i = 0; i < R; i++)
        for(ll j = 0; j < C; j++)
            cin >> G[i][j];
    // Oznacme vsetky policka ako nenavstivene
    for(ll i = 0; i < R; i++)
        for(ll j = 0; j < C; j++)
            vis[i][j] = false;
    vis[0][0] = true;
    // Zaujima nas odpoved pre lave horne policko
    cout << go(0, 0) << '\n';

    return 0;
}
```

Z hrubej sily vzorák?

Ako sme už povedali, niekedy stačí malé pozorovanie, aby sme z riešenia hrubou silou spravili vzorák.

Skúsme si pre každé políčko zrátať, koľko by sme zarobili, keby sme na ňom začínali, ale nemohli stúpiť na políčko $[r, c]$.

Naša nová rekurzia si okrem súradníc políčka, na ktorom stojíme, pamätá aj to, na ktoré políčko nesmieme stúpiť. Nepotrebujeme teda už veľké dvojrozmerné pole navštívených políčok. Totiž, ak si pamätáme iba to, na ktorom políčku sme boli bezprostredne predtým, ako sme sa dostali na to aktuálne, tak určite na žiadne políčko nestúpime dvakrát. Je to kvôli tomu, že nemôžeme chodiť doľava. Rozmyslite si, že nám táto informácia naozaj stačí.

Dokonca nepotrebujeme ani súradnice predchádzajúceho políčka. Stačí nám vedieť smer, z ktorého sme prišli, čo si môžeme reprezentovať napríklad číslami 1, 2, 3. Naša rekurzia má teda stav `riadok, stlpec, odkial`.

Všimnime si teraz veľmi užitočnú vec. Ak už niekedy zrátame odpoveď pre nejaký stav `riadok, stlpec, odkial`, tak už nikdy ho nemusíme rátať znova. Stačí, ak si zapamätáme výsledok a keď ho budeme potrebovať zrátať nabudúce, iba sa pozrieme na ten výsledok, ktorý máme uložený.

Vnútro našej rekruzívnej funkcie teda bude vyzerať asi tak, že sa najskôr pozrieme, či už nie sme na pravom dolnom políčku. Ak áno vrátime ako odpoveď 0. Ak nie sme, tak sa pozrieme, či už nemáme výsledok pre aktuálny stav uložený. Ak áno, tak ho vrátime. Ak ho nemáme uložený, tak sa rekurzívne zavoláme do tých dvoch, alebo troch smerov, do ktorých môžeme. Skúšame totiž 3, ale z jedného sme možno prišli, takže tam nepôjdeme. Nakoniec si iba zapamätáme odpoveď pre aktuálny stav a vrátime ju.

Na konci iba vypíšeme odpoveď pre stav `0, 0, -1`.

Akú to má zložitosť? Každý stav zrátame v $O(1)$, pretože sa iba dvakrát rekurzívne zavoláme. Stavov je $R \cdot C \cdot 3$. Celková časová zložitosť teda bude $O(RC)$. Pamätáme si vstup a odpovede pre všetky stavy, takže aj pamäť bude $O(RC)$.

Jeden problém, ktorý sa môže vyskytnúť je hĺbka rekurzie. Môže nám totiž dôjsť miesto na zásobníku. To má ale jednoduché riešenie. Nezavoláme sa hneď na výsledný stav, ale budeme sa postupne volať od posledných stavov až k tomu, ktorý nás najviac zaujíma.

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

const ll MAXR = 1111, MAXC = 1111;
ll R, C;
ll G[MAXR][MAXC];
ll memo[3][MAXR][MAXC];
```

```
enum {LEFT, TOP, BOTTOM};

ll go(ll from, ll r, ll c);

ll go_up(ll r, ll c) {
    if(r > 0)
        return G[r - 1][c] + go(BOTTOM, r - 1, c);
    return INT_MIN;
}

ll go_down(ll r, ll c) {
    if(r < R - 1)
        return G[r + 1][c] + go(TOP, r + 1, c);
    return INT_MIN;
}

ll go_right(ll r, ll c) {
    if(c < C - 1)
        return G[r][c + 1] + go(LEFT, r, c + 1);
    return INT_MIN;
}

ll go(ll from, ll r, ll c) {
        // Ak sme vpravo dole, tak je odpoved 0
    if(r == R - 1 && c == C - 1)
        return 0;
        // Ak sme tento stav uz niekedy zratali, tak iba vratime jeho vysledok,
        // nemusime ho pocitat znova
    if(memo[from][r][c] != -1)
        return memo[from][r][c];
    ll ans = INT_MIN;
        // Ak sme prisli zlava, mozeme ist aj hore, aj dole, aj doprava
    if(from == LEFT)
                ans = max({ans, go_up(r, c), go_down(r, c), go_right(r, c)});
        // Ak sme prisli zhora, tak mozeme ist iba dole, alebo doprava
    else if(from == TOP)
                ans = max({ans, go_down(r, c), go_right(r, c)});
        // Ak sme prisli zdola, tak mozeme ist iba hore, alebo doprava
    else
                ans = max({ans, go_up(r, c), go_right(r, c)});
        // Zapamatame si odpoved pre tento stav a hned ju aj vratime
    return memo[from][r][c] = ans;
}

int main() {
    cin >> R >> C;
    for(ll i = 0; i < R; i++)
        for(ll j = 0; j < C; j++)
            cin >> G[i][j];

        // Vytvaranie pola pre zapamatavanie vysledkov uz zratanych stavov
        // Ak je na policku memo[riadok][stlpec][odkial] hodnota -1, tak sme
        // tento stav este nezratali
    for(ll i = 0; i < 3; i++)
        for(ll j = 0; j < MAXR; j++)
            memset(memo[i][j], -1, sizeof(memo[i][j]));

        // Kvoli hlbke rekurzie ratame od poslednych stavov
    for(ll col = C - 1; col >= 0; col--) {
        for(ll row = R - 1; row >= 0; row--) {
            go(LEFT, row, col);
            go(TOP, row, col);
            go(BOTTOM, row, col);
        }
    }

    cout << go(TOP, 0, 0) << '\n'; // Mozeme dat TOP, lebo urcite nepojdeme na zaciatku hore


    return 0;
}
```

Listing programu (Python)

```
import sys

sys.setrecursionlimit(100000)
MAXR = 1111
MAXC = 1111
INF = 10**9

R, C = [int(x) for x in input().split()]
vis = [[False for _ in range(C)] for _ in range(R)]
G = [[int(x) for x in input().split()] for _ in range(R)]
memo = [[[-1 for _ in range(C)] for _ in range(R)] for _ in range(3)]

LEFT, TOP, BOTTOM = 0, 1, 2


def go_up(r, c):
    if r > 0:
        return G[r - 1][c] + go(BOTTOM, r - 1, c)
    return -INF
```

```
def go_down(r, c):
    if r < R - 1:
        return G[r + 1][c] + go(TOP, r + 1, c)
    return -INF


def go_right(r, c):
    if c < C - 1:
        return G[r][c + 1] + go(LEFT, r, c + 1)
    return -INF


def go(prev, r, c):
    # Ak sme uz vpravo dole, odpoved je 0
    if r == R - 1 and c == C - 1:
        return 0
    # Ak sme tento stav uz ratali, tak iba vratime jeho vysledok,
    # nemusime ho pocitat znova
    if memo[prev][r][c] != -1:
        return memo[prev][r][c]
    ans = -INF
    # Ak sme prisli zlava, mozeme ist aj hore, aj dole, aj doprava
    if prev == LEFT:
        ans = max([ans, go_up(r, c), go_down(r, c), go_right(r, c)])
    # Ak sme prisli zhora, mozeme ist iba dole, alebo doprava
    elif prev == TOP:
        ans = max([ans, go_down(r, c), go_right(r, c)])
    # Ak sme prisli zdola, mozeme ist iba hore, alebo doprava
    else:
        ans = max([ans, go_up(r, c), go_right(r, c)])
    # Zapamatame si vysledok pre tento stav, aby sme ho uz nabuduce
    # nemuseli pocitat znova
    memo[prev][r][c] = ans
    return ans

# Ideme od poslednych stavov, aby sme neminuli miesto na stacku kvoli rekurzii
for col in range(C - 1, -1, -1):
    for row in range(R - 1, -1, -1):
        go(LEFT, row, col)
        go(TOP, row, col)
        go(BOTTOM, row, col)

print(go(TOP, 0, 0))
```

Technika zapamätávania si už zrátaných výsledkov sa zvykne označovať ako dynamické programovanie. My sme tu použili jeho rekurzívnu formu, ktorá sa často nazýva rekurzia s memoizáciou. Samozrejme, táto úloha sa dá riešiť rovnako jednoducho aj iteratívne, rekurzia je však pre množstvo ľudí o niečo intuitívnejšia.

Marek

4. Ohromná estetickosť

(max. 12 b za popis, 8 b za program)

V prvom rade si uvedomme, čo je našou úlohou.

Zistite, koľko najviac si ich Matúš môže priniesť domov, ak chce v obchode stráviť iba jeden súvislý časový úsek a chce, aby množina týchto gúľ spĺňala zaujímavú ohromne estetickú podmienku: Všetky získané gule musí byť možné zavesiť vedľa seba tak, aby vzniknutá postupnosť ich farieb bola rovnaká z ľavej aj z pravej strany.

V preklade toto znamená, že chceme nájsť najdlhší súvislý podreťazec, z ktorého vieme spraviť palindróm. *Spraviť palindróm* znamená, že môžeme meniť poradie písmen v tomto podreťazci tak, aby výsledný reťazec bol palindróm.

Čo vieme o palidróme?

Palindróm je reťazec, ktorý je rovnaký z jednej aj z druhej strany. Existuje v ňom teda nejaký stred, od ktorého sa vľavo a vpravo nachádzajú rovnaké znaky v rovnakom poradí.

Keďže všetky znaky, nachádzajúce sa vľavo od stredu, sa nachádzajú aj vpravo od stredu, vieme, že sa v palindróme každý znak, nachádzajúci v jednej polovici, nachádza celkovo dvakrát.

Samozrejme, to platí, ak má palindróm párnu dĺžku a jeho stred je tak presne medzi dvoma písmenami. Ak má palindróm nepárny počet znakov, jeden znak je presne v strede. Tento stredný znak sa nenachádza vľavo ani vpravo od stredu, preto nemá svoju "dvojičku" v druhej polovici palindrómu a je nám jasné, že sa v palindróme nachádza práve raz.

Z takejto úvahy sa dozvedáme, že v palindróme párnej dĺžky sa musí každý znak vyskytovať párny-počet-krát a v palindróme nepárnej dĺžky sa musia všetky znaky okrem práve jedného vyskytovať párny-počet-krát.

Zjednodušene: V palindróme sa musí žiadny alebo jeden znak vyskytovať párny-počet-krát.

Všetky podreťazce

Okay. Chceme asi zistiť početnosti znakov v jednotlivých podreťazcoch vstupného reťazca gúľ a nájsť taký najdlhší podreťazec, že žiadny alebo práve jeden znak má v ňom nepárnu početnosť.

Ak má reťazec dĺžku n, existuje $\frac{(n+1)\cdot n}{2}$ jeho podreťazcov (1 dlžky n, 2 dĺžky $n-1$, 3 dĺžky $n-2$, $\ldots$ n dĺžky 1), čo je asymptoticky n^2. Ak by sme pre chceli každý jeden podreťazec rátať početnosti jeho znakov osobitne, museli by sme spracovať vštky znaky vo všetkých podreťazcoch. Počet znakov vo všetkých podreťazcoch je asymptoticky n^3.

Okrem toho by sme pre každý podreťazec museli skontrolovať k početností znakov, aby sme zistili, či spĺňa podmienky. k je veľkosť abecedy, no v zadaní sme sa dozvedeli, že je to najviac 20, preto môžeme toto číslo brať ako konštantu. Z toho vidíme časovú zložitosť $\mathcal{O}(n^3)$.

Niečo lepšie

Predchádzajúce riešenie vieme značne vylepšiť jednoduchou myšlienkou. Nepotrebujeme pre každý jeden podreťazec rátať početnosť odznova, vieme využiť informáciu z o jedno menšieho podreťazca.

Ak nebudeme rátať početnosti znakov v každom podreťazci osobitne, ale využijeme to, že početnosť v reťazci o dĺžke x sa len pre jeden znak líši od početnosti jeho podreťazca o dĺžke $x-1$. To znamená, že počas výpočtu početností pre nejaký podreťazec so znakmi C_i až C_j sme už vyrátali aj početnosti pre podreťazce so znakmi $C_i \to C_{i+1}, C_i \to C_{i+2}, \ldots C_i \to C_{j-1}$. To znamená, že nám vlastne stačí spracovať len n podreťazcov $(C_0 \to C_n, C_1 \to C_n, \ldots C_n \to C_n)$. Priemerná dĺžka týchto podreťazcov je $\frac{n}{2}$ znakov.

Vidíme, že takýmto prístupom nám časová zložitosť klesne na $\mathcal{O}(n^2)$. Pamäťová zložitosť je tu $\mathcal{O}(n)$. Potrebujeme si pamätať len celý vstupný reťazec.

Zjednodušenie a prefixy

Položme si otázku: Naozaj potrebujeme vedieť konkrétne početnosti znakov? Hm... Povedal by som, že nás aj tak vo výsledku zaujímajú iba parity týchto početností (či sú párne alebo nepárne). To by ale predchádzajúce riešenie aj tak nezlepšilo, aj tak by sme museli spracovať n podreťazcov.

Navyše, prečo musíme spracovať osobitne n podreťazcov? Teraz si uvedomme, ak máme informácie o paritách v podreťazcoch $C_i \to C_j$ a $C_i \to C_k$ pre $j < k$, vieme z toho zistiť aj paritu podreťazcu $C_j \to C_k$. V podstate prefixové súčty.

Ak je medzi indexami i a j početnosť nejakého písmena a_{ij} a medzi indexami i a k a_{ik}, početnosť mezi indexami j a k je potom $a_{jk} = a_{ik} - a_{ij}$. To isté platí aj pre paritu tejto početnosti. Ak, napríklad, $a_{ij} = parne$ a $a_{ij} = neparne$, tak $a_{jk} = neparne$.

Zo spracúvania n podreťazcov, začínajúcich na indexoch $0 \ldots n-1$, sa dostávame k tomu, že nám stačí spracovať celý reťazec len raz. Počas jeho spracúvania, spracúvame postupne všetky podreťazce začínajpce na indexe 0. V predchádzajúcom odesku sme si ukázali, že ak poznáme parity reťazcov, začínajúcich na rovnakom indexe, poznáme aj parity medzi ich koncami. Ak poznáme parity všetkých podreťazcov, začínajúcich na indexe 0, poznáme aj parity všetkých podreťazcov medzi všetkými ich koncami, čo sú vlastne všetky podreťazce celého reťazcu. Čo s nimi?

Xor a zapamätanie najľavejšej

$a_{jk} = neparne$, ak $a_{ij} \neq a_{ij}$. To vyzerá ako xor.

V zadaní sa spomína, že v reťazci je najviac 20 rôznych znakov. Každý z týchto 20 znakov môže mať párnu alebo nepárnu paritu. To je dokopy 2^{20} kombinácií parít jednotlivých znakov.

Hm... Informáciu o parite nejakého podreťazcu si môžeme reprezentovať, napríklad, nejakým binárnym číslom s 20 bitmi. Ak máme dva takéto podreťazce, začínajúce na rovnakom indexe, xorom ich parít dostaneme paritu podreťazcu medzi ich koncami.

Ku každej parite existuje 20 takých, ktoré sa líšia práve v jednom znaku. Ak spracúvame vstupný reťazec z ľava do prava a sme na nejakom i-tom znaku reťazca, zaujíma nás, či a kde vľavo od neho sa nachádza znak, na ktorom sa parita nelíši od aktuálnej alebo sa líši v práve jednom znaku. To znamená, že podreťazec medzi nimi má vyhovujúcu paritu (0 alebo 1 nepárnych).

Dokonca, môžeme povedať, že má zmysel hľadať len tú najľavejšiu z takých parít. Hľadáme *najdlhší* podreťazec, preto pre nás nemá zmysel nie-najľavejšia parita, pretože tá by určite nepatrila najdlhšiemu takému podreťazcu, ak by existovala nejaká ľavejšia, čiže vzdialenejšia od aktuálne spracúvaného znaku.

Zhrnutie vzorového riešenia

Pamätáme si pre každú paritu, kde najskôr sa v reťazci vyskytuje. Pre každý aktuálne spracúvaný index reťazcu si nájdeme najľavejšiu vyhovujúcu paritu (ak existuje), lebo tá začína najdlhší podreťazec končiaci v aktuálnom znaku. Keď takto spracujeme celý reťazec, je nám známa dĺžka najdlhšieho podreťazcu, z ktorého je možné vytvoriť palindróm.

Časová zložitosť. Pre každý znak v reťazci potrbujeme len nájsť najľavejší s vyhovujúcou paritou, ktorých je $k+1$. To nám dáva krásne $\mathcal{O}(n \cdot k)$, kde n je dĺžka reťazcu a k je počet farieb. Ak si ale povieme, že 20 (najviac farieb) je konštanta, ostane nám $\mathcal{O}(n)$.

S pamäťou je to tu trošičku smutnejšie. Okrem vstupu si ešte potrebujeme pamätať najľavejšiu pozíciu pre každú možnú paritu. Tých je až 2^{20}, takže pamäťová zložitosť nám tu vychádza na $\mathcal{O}(n + 2^{20})$.

Listing programu (C++)

```
#include <cstdio>

using namespace std;

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

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

int main () {
        // i-ty bit v prefix_parity hovori, ci je pocet i-tej farby neparny
        int prefix_parity = 0;

        // kde sa najskor nachadza prva takato "parita"?
        int index_of_first[1 << 20];

        // dajme tam najprv velke cisla
    for (int i = 1; i < 1 << 20; i++)
        index_of_first[i] = 300005;

    // ziadne sa nachadza najskor pred prvym pismenom
    index_of_first[0] = -1;

    int n;
        char input[300003];
    scanf("%d\n%s", &n, input);

    // najprv mame najdeny najdlhsi usek o dlzke 0
        int result = 0;

        // podme po pismenkach vo vstupe
    for (int here = 0; here < n; here++) {
        // prixorujeme here-te pismeno
        prefix_parity ^= (1 << (input[here] - 'a'));
        // je usek s aktualnou paritou so ziadnym neparnym poctom dlhsi?
        result = max(result, here - index_of_first[prefix_parity]);

        // je nejaky usek s aktualnou paritou s jednym neparnym poctom dlhsi?
        for (int color = 1; color < (1 << 20); color <<= 1)
                result = max(result, here - index_of_first[prefix_parity ^ color]);

        // ak sme este taku paritu nevideli, here je jej prvy/najskorsi vyskyt
        index_of_first[prefix_parity] = min(index_of_first[prefix_parity], here);
    }

    // mame to
    printf("%d\n",result);

    return 0;
}
```

MichalS

5. Všetko so všetkým súvisí

(max. 12 b za popis, 8 b za program)

Asi prvá vec, ktorú si všimneme, je, že hľadané číslo bude deliteľné číslom n, ktoré sme dostali na vstupe. Tu sa nám črtá prvé riešenie, ktoré jednoducho vyskúša všetky násobky čísla n a vypíše prvý, ktorý má ciferný súčet rovný n.

Výsledné číslo musí mať aspoň $\frac{n}{9}$ cifier, a takých čísel je $10^{\frac{n}{9}}$.

Z nich každé n-té je deliteľné n, čiže toto riešenie skúša exponenciálne veľa možností. Keďže výsledok môže obsahovať aj nuly, nemáme žiadny horný limit na dĺžku hľadaného čísla, takže tento program môže bežať dokonca ľubovoľne dlho. Poďme teda nájsť lepšie riešenie.

Vcelku dobrý nápad vyzerá byť generovanie hľadaného čísla postupne cifru po cifre. Budeme to robiť zľava doprava, teda od najvýznamnejších číslic k najmenej významným. Pozrime sa na to, čo sa stane, keď k nejakému číslu m s ciframi $c_1 c_2 \ldots c_k$ pripíšeme na koniec cifru d a dostaneme tak číslo $c_1 c_2 \ldots c_k d$. Jeho ciferný súčet sa zvýši o d. Zároveň sa už nikdy neskôr nemôže znížiť, keďže všetky cifry sú nezáporné. Zaujímavé je, čo sa stane s jeho zvyškom. Pri pripísaní cifry na koniec čísla sme všetky cifry posunuli o jedno miesto doľava, čo zodpovedá vynásobeniu desiatimi a napokon sme pričítali cifru d. Hodnota nového čísla je tak $10m + d$.

Teraz si uvedomme, že pri skúmaní deliteľnosti nás vlastne až tak nezaujíma samotné číslo m, stačí nám poznať jeho zvyšok po delení n, označme ho z. Ten sa pripísaním cifry d zmení na $(10z + d) \bmod n$.

Našu úlohu si tak vieme previesť na prehľadávanie grafu, kde každý vrchol reprezentuje číslo, ktoré máme aktuálne napísané, a každá (jednosmerná) hrana reprezentuje pripísanie číslice na koniec čísla. Samotné napísané

čísla si však pamätať nechceme, keďže sú príliš dlhé. O každom takomto čísle nám stačí pamätať si jeho ciferný súčet a jeho zvyšok po delení n.

Na začiatku máme napísaný prázdny reťazec cifier, ktorý má ciferný súčet 0 a zvyšok tiež 0 (pretože keď napíšeme cifru d, dostaneme zvyšok $10 \cdot 0 + d = d \bmod n$). K tomuto prázdnemu reťazcu teraz budeme pripisovať nejaké cifry, čo v tomto poňatí znamená nejaké presuny po hranách. Nakoniec sa chceme dostať do stavu s ciferným súčtom n a zvyškom 0.

Na prehľadanie grafu použijeme prehľadávanie do šírky (BFS). Na začiatku si do prázdnej fronty vložíme počiatočný stav (ciferný súčet aj zvyšok rovné 0). Postupne vyťahujeme z fronty nové stavy a ku každému skusíme pridať každú cifru. Zo stavu so súčtom s a zvyškom z sa cifrou d dostaneme k súčtu $s + d$ a zvyšku $(10 \cdot z + d) \bmod n$. Ku každému stavu si tiež pamätáme, odkiaľ a s akou cifrou sme sa doň dostali, aby sme vedeli spätne zrekonštruovať vytvorené číslo. Ak nájdeme stav, ktorý má rovnaký ciferný súčet aj zvyšok, ako sme už videli, tento stav nebudeme skúmať znova (je to ten istý vrchol grafu).

Na konci programu pomocou zapamätaných spätných liniek vytvoríme hľadané číslo po jednotlivých cifrách. Treba mať na pamäti, že toto číslo dostaneme odzadu, takže ho pred výpisom musíme ešte otočiť.

Zadanie však od nás chce, aby sme našli najmenšie možné číslo, nie ľubovoľné. Tu je dôležité, že sme použili BFS. Ukážeme, že tento algoritmus našiel naozaj najmenšie možné číslo. BFS nájde cestu s najmenším počtom hrán, teda dostaneme číslo s najmenším počtom cifier (čo hrana, to cifra). Najskôr spracúvame stavy s jednou napísanou cifrou, potom s dvomi, tromi atď. Tiež je dôležité spracúvať hrany idúce z jedného vrcholu v poradí podľa pripísanej cifry od 0 do 9. Spomedzi rovnako dlhých čísel sa tak vždy dostane menšie číslo vo fronte pred väčšie. Ak dve čísla začínajú rovnako a líšia sa len v poslednej cifre, potom číslo s menšou poslednou cifrou bude vo fronte skôr. Ak sa rovnako dlhé čísla líšia skôr ako v poslednej cifre, potom začiatok (bez poslednej cifry) menšieho sme vytiahli z fronty skôr, a tak všetky čísla, ktoré vzniknú z neho, sa dostanú do fronty skôr a usporiadanosť čísel vo fronte sa zachová. Stavy teda spracúvame v poradí od najmenšieho napísaného čísla k väčším, ale tieto čísla v pamäti nedržíme, každý stav reprezentujeme len dvojicou čísel – ciferný súčet a zvyšok.

Uvažujeme iba ciferné súčty menšie alebo rovné n a existuje n rôznych zvyškov po delení n, máme teda $O(n^2)$ stavov (vrcholov grafu), do ktorých sa vieme dostať. V každom stave máme 10 možností, akú cifru pripísať, teda z každého stavu ide 10 (konštantne veľa) hrán. Časová zložitosť prehľadavánia je teda $O(n^2)$, pamäťová je rovnako $O(n^2)$.

Listing programu (Python)

```
from collections import deque


def find_number(n):
    #came je dvojrozmerne pole, v ktorom si drzime graf roznych stavov
    #a informacie o nich
    #came[ciferny sucet][zvysok modulo n] =
    #   (ciferny sucet v predoslom stave,
    #   zvysok modulo n v predoslom stave,
    #   cifra, ktorou sme sa sem dostali)
    #None znamena, ze tento stav sme este nevideli
    came = [[None] * n for i in range(n + 1)]
    came[0][0] = (0, 0, 0)

    #Vytvorime si frontu a vlozime do nej pociatocny stav
    #V pythone je fronta uz implementovana v moduli collections ako deque
    #Na zaciatku je ciferny sucet aj zvysok 0
    q = deque()
    q.append((0, 0))

    #Prehladavame graf stavov do sirky
    while q:
        digit_sum, remainder = q.popleft()
        #Vyskusame vsetky mozne cifry
        for d in range(10):
            #spocitame novy zvysok a pozrieme sa, ci sme uz taky stav videli
            new_remainder = (10 * remainder + d) % n
            if digit_sum + d <= n and came[digit_sum + d][new_remainder] is None:
                came[digit_sum + d][new_remainder] = (digit_sum, remainder, str(d))
                q.append((digit_sum + d, new_remainder))

    #Spatne zrekonstruujeme hladane cislo
    digit_sum, remainder = n, 0
    digits = []
    while digit_sum > 0:
        #Pohneme sa o jednu cifru naspat podla informacii zapamatanych v came
        digit_sum, remainder, digit = came[digit_sum][remainder]
        digits.append(digit)

    #Ziskali sme ho odzadu, takze ho este musime otocit
    return "".join(reversed(digits))


print(find_number(int(input())))
```

mišof

6. Interpol

(max. 12 b za popis, 8 b za program)

Zamyslime sa najskôr nad statickou verziou tejto úlohy: ak máme daných n úsekov cesty $[x_i, y_i]$, ako vyzerá ich prienik? Kedy je neprázdny?

Aby nejaký bod x patril do prieniku všetkých intervalov, museli už všetky začať, teda musí platiť $x \geq \max_i x_i$. Tiež musí platiť, že žiaden interval ešte nesmel skončiť, teda $x \leq \min_j y_j$.

Z toho vidíme, že môžu nastať dva prípady: ak $\max_i x_i > \min_j y_j$, prienik je prázdny – niektorý interval skončí skôr ako niektorý iný začne. V opačnom prípade je prienik neprázdny a je ním zjavne práve uzavretý interval $[\max_i x_i, \min_j y_j]$.

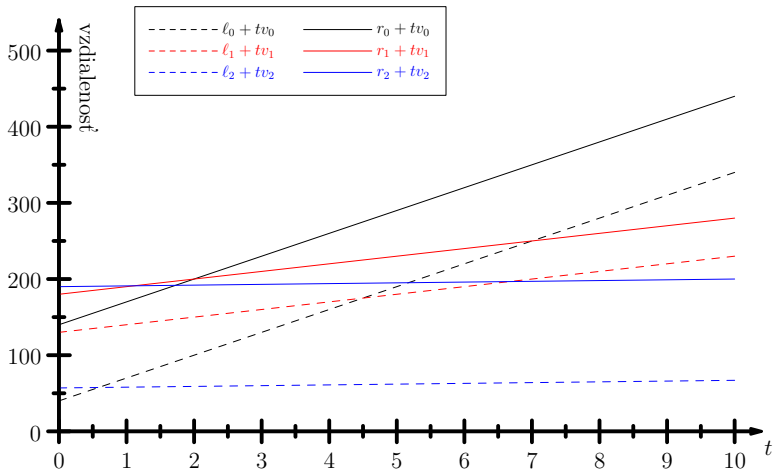
Dĺžka intervalu ako funkcia

Ak si v našej úlohe zvolíme nejaký konkrétny okamih $t \geq 0$, vieme si vypočítať, ako v danom okamihu vyzerajú slepé intervaly: i-ty z nich bude $[\ell_i + tv_i, r_i + tv_i]$.

Nás zaujíma, akú najdlhšiu dĺžku (a pre aké t) bude mať tento interval. Uvažujme preto nasledujúcu funkciu: $f(t) = \min_j(r_j + tv_j) - \max_i(\ell_i + tv_i)$. Zjavne platí, že ak $f(t) < 0$, tak je v čase t prienik prázdny, a inak $f(t)$ udáva jeho dĺžku. Našu úlohu teda vieme preformulovať tak, že hľadáme, pre ktoré $t \geq 0$ nadobúda funkcia f svoje maximum.

Príklad funkcie f

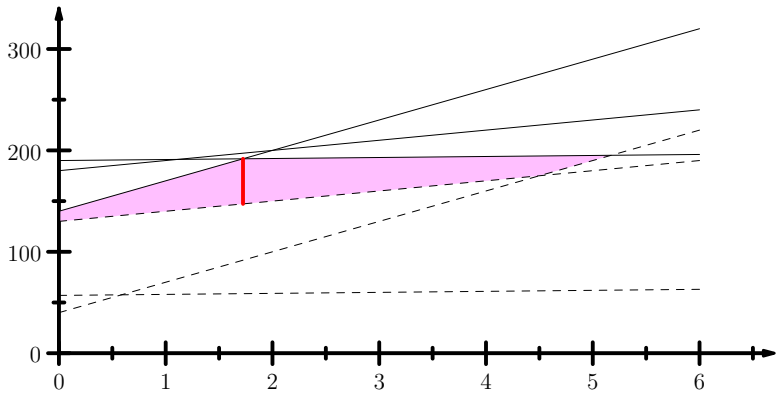
Na prvom grafe nižšie vidíme, ako sa tri rôzne slepé intervaly pohybujú v čase. Na x-ovej osi je čas, na y-ovej vzdialenosť na ceste. Znázornené intervaly zodpovedajú poslednému príkladu v zadaní, až na to, že ℓ_2 je o čosi väčšie, aby to lepšie vyzeralo.



Čiarkované čiary predstavujú začiatky jednotlivých intervalov. Nás vždy zaujíma posledný začiatok (hodnota $\max_i(\ell_i + tv_i)$), čiže najvyššie sa nachádzajúca čiarkovaná čiara.

Analogicky nás v každom čase zaujíma prvý koniec intervalu, čiže najnižšie sa nachádzajúca plná čiara. Ak sa tá nachádza nad všetkými čiarkovanými, intervaly majú v danom čase neprázdny prienik.

Na druhom grafe sme vyfarbili oblasť, ktorá zodpovedá neprázdnemu prieniku intervalov, a hrubou čiarou sme vyznačili optimálne riešenie našej úlohy.



Dôležitá vlastnosť funkcie f

V čase $t = 0$ vidíme, že najvyššia čiarkovaná čiara zodpovedá intervalu 1, zatiaľ čo najnižšia plná čiara intervalu 0. No a keďže interval 0 sa hýbe rýchlosťou 30 a interval 1 len rýchlosťou 10, keď teraz pôjdeme v čase ďalej, bude dĺžka prieniku plynule rásť (rýchlosťou $30 - 10 = 20$ za jednotku času).

V čase $t = 50/29$ sa spodnou plnou čiarou stane čiara zodpovedajúca intervalu 2. Ten sa hýbe len rýchlosťou 1. Od tohto okamihu ďalej teda bude čiarkovaná čiara „dobiehať" plnú. Dĺžka prieniku sa teda bude meniť o $1 - 10 = -9$ za jednotku času.

V čase $t = 9/2$ nastane ďalšia zmena: hornou čiarkovanou čiarou sa stane čiara zodpovedajúca intervalu 0. Od tejto chvíle sa dĺžka prieniku mení rýchlosťou $1 - 30 = -29$ za jednotku času, a čoskoro už prienik prestane existovať.

Na tomto príklade si môžeme všimnúť všeobecné pravidlo, ktoré bude vždy platiť: rýchlosť, ktorou rastie dĺžka prieniku, sa môže s rastúcim časom *len zmenšovať*. Totiž vždy, keď nastane zmena najvrchnejšej čiarkovanej čiary, tá nová rastie rýchlejšie, a vždy, keď nastane zmena najspodnejšej plnej čiary, tá nová rastie od tej predchádzajúcej pomalšie.

Keďže na začiatku môže byť rýchlosť rastu kladná, znamená to, že samotná *dĺžka prieniku* (presnejšie, hodnota funkcie f, ktorá nás zaujíma) vo všeobecnosti *najskôr nejaký čas rastie, potom je nejaký čas konštantná, a nakoniec klesá*.

My teraz potrebujeme nájsť maximum takejto funkcie. Spravíme to binárnym vyhľadávaním.[1]

Binárne vyhľadávanie

V ľubovoľnom čase t si vieme v čase $O(n)$ nájsť aj najspodnejšiu plnú čiaru, aj najvrchnejšiu čiarkovanú, a u oboch sa pozrieť na to, ako rýchlo rastú. Rozdiel týchto dvoch hodnôt nám povie, či v danom bode funkcia f ešte rastie, je práve konštantná, alebo už klesá.

Začneme tým, že sa pozrieme na čas $t = 0$. Ak už v tomto okamihu f nerastie, je $f(0)$ odpoveďou, ktorú hľadáme a môžeme skončiť.

Označme teraz m najväčšie číslo na vstupe (či už ide o súradnicu alebo rýchlosť). Pripomíname, že pre testovacie dáta platilo $m \leq 10^6$. Potom tvrdíme, že po čase m už nenastane žiadna zmena: od tohto času ďalej musí byť aj čiarkovaná aj plná čiara stále tá istá. Totiž ak iná bola od nej rýchlejšie/pomalšie rastúca, tak na začiatku mala táto pred ňou „náskok" najviac m, a keďže sú všetky čísla celé, tá druhá čiara tento náskok dobiehala aspoň rýchlosťou 1 za jednotku času.

(Algebraicky, priesečník priamok $a + bt$ a $c + dt$, kde $c \neq d$, je v bode $t = (a - c)/(d - b)$, no a v základnom tvare má tento zlomok má čitateľ $\leq m$ a menovateľ ≥ 1.)

Máme teraz dve pozorovania: v čase $t_{lo} = 0$ funkcia f ešte rastie, zatiaľ čo v čase $t_{hi} = m$ už určite nerastie. Od tohto okamihu ďalej môžeme použiť spomínané binárne vyhľadávanie: dokola opakujeme, že sa pozrieme do stredu intervalu, vyhodnotíme, či tam f ešte rastie alebo už nerastie, a podľa toho posunieme buď t_{lo} alebo t_{hi}.

Trocha technických detailov na záver

Pri praktickej implementácii si treba dať pozor na zaokrúhľovacie chyby. Hodnotu optimálneho t, a teda aj hodnotu $f(t)$, zistíme len približne. Zadanie síce sľubuje, že testovač je tolerantný, ale aj tak ostáva jeden okrajový prípad, na ktorý si treba dať pozor: ak je optimálna odpoveď presne $f(t) = 0$, ale správne t netrafíme presne, dostaneme ako $f(t)$ hodnotu, ktorá je tesne pod nulou. Ak vtedy prehlásime, že riešenie neexistuje, dáme nesprávnu odpoveď.

V jednom autorskom riešení sme pre istotu celý záver riešenia spravili exaktne: po dostatočnom počte iterácií binárneho vyhľadávania máme nájdené nejaké t. Okrem tohto t si exaktne (ako zlomok dvoch veľkých čísel) dopočítame najbližšie menšie aj najbližšie väčšie t, v ktorom sa mení niektorá hraničná čiara, a v oboch časoch exaktne vyhodnotíme funkciu f. Takto máme istotu, že sme určite našli optimálne riešenie.

Na úspešné vyriešenie úlohy však stačilo aj použitie reálnych (floating-point) čísel a vhodné zaokrúhľovanie. Správny čas t, aj správna odpoveď je totiž vždy zlomok, ktorého menovateľ je najviac m. V našom prípade to teda znamená, že ak je maximum f naozaj záporné, tak je vždy menšie alebo rovné -10^{-6}. A teda ak nám vychádza odpoveď výrazne bližšia nule, môžeme si byť rozumne istí, že správnou odpoveďou je samotná nula.

Listing programu (C++)

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

[1] Existuje aj korektné riešenie založené na tzv. ternárnom vyhľadávaní, ktoré funguje trochu ináč. Značnú časť riešenia však majú oba tieto prístupy podobnú, preto sme sa rozhodli popísať len binárne vyhľadávanie. K ternárnemu uvedieme na konci len program.

```
int N;
vector<int> L, R, V;

long double get_length(long double t) {
    long double maxlo = -1e13, minhi = 1e13;
    for (int n=0; n<N; ++n) {
        maxlo = max( maxlo, L[n] + t*V[n] );
        minhi = min( minhi, R[n] + t*V[n] );
    }
    return minhi - maxlo;
}

bool does_still_grow(long double t) {
    long double maxlo = -1e13, minhi = 1e13;
    int lospeed = -1, hispeed = -1;
    for (int n=0; n<N; ++n) {
        long double curlo = L[n] + t*V[n];
        if (curlo > maxlo) { maxlo = curlo; lospeed = V[n]; }
        long double curhi = R[n] + t*V[n];
        if (curhi < minhi) { minhi = curhi; hispeed = V[n]; }
    }
    return hispeed > lospeed;
}

void print_answer(long double t) {
    long double answer = get_length(t);
    if (answer < -1e-9) cout << "-1\n"; else cout << fixed << setprecision(10) << answer << endl;
    exit(0);
}

int main() {
    cin >> N;
    L.resize(N); R.resize(N); V.resize(N);
    for (int n=0; n<N; ++n) cin >> L[n] >> R[n] >> V[n];

    if (!does_still_grow(0)) print_answer(0);

    long double lo = 0, hi = 1234567;
    for (int loop=0; loop<120; ++loop) {
        long double mid = (lo + hi) / 2;
        if (does_still_grow(mid)) lo = mid; else hi = mid;
    }
    print_answer(hi);
}
```

Listing programu (C++)

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

int N;
vector<int> L, R, V;

long double get_length(long double t) {
    long double maxlo = -1e13, minhi = 1e13;
    for (int n=0; n<N; ++n) {
        maxlo = max( maxlo, L[n] + t*V[n] );
        minhi = min( minhi, R[n] + t*V[n] );
    }
    return minhi - maxlo;
}

int main() {
    cin >> N;
    L.resize(N); R.resize(N); V.resize(N);
    for (int n=0; n<N; ++n) cin >> L[n] >> R[n] >> V[n];

    long double lo = 0, hi = 1234567;
    for (int loop=0; loop<120; ++loop) {
        long double m1 = (lo+lo+hi)/3, m2 = (lo+hi+hi)/3;
        long double len1 = get_length(m1), len2 = get_length(m2);
        if (len1 < len2) lo = m1; else hi = m2;
    }
    long double answer = get_length( (lo+hi)/2 );
    if (answer < -1e-9) cout << "-1\n"; else cout << fixed << setprecision(10) << answer << endl;
}
```

7. Chicago

nulano
(max. 12 b za popis, 8 b za program)

Hrubá sila

Riešenie hrubou silou je jednoduché: Vyskúšame všetky možné umiestnenia dvojíc vysielačov, a pre každé spočítame počet pokrytých obyvateľov. Keďže Chicago má n^2 mrakodrapov, možných rozložení dvojice vysielačov je $O(n^4)$. Ak potom pre každé rozloženie zrátame v konštantnom čase počet pokrytých obyvateľov pomocou predpočítaných súčtov riadkov a stĺpcov, môžeme získať 4 body za riešenie s časovou zložitosťou $O(n^4)$.

Listing programu (C++)

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0); cout.tie(0);

    int n; cin >> n;

    vector<vector<int>> d(n, vector<int>(n));
    vector<long long> riadky(n), stlpce(n);
    for (int y = 0; y < n; y++) {
        for (int x = 0; x < n; x++) {
            cin >> d[y][x];
            riadky[y] += d[y][x];
            stlpce[x] += d[y][x];
        }
    }

    long long best = 0;

    for (int y1 = 0; y1 < n; y1++) {
        for (int x1 = 0; x1 < n; x1++) {
            long long s1 = riadky[y1] - d[y1][x1] + stlpce[x1] - d[y1][x1];
            for (int y2 = y1; y2 < n; y2++) {
                for (int x2 = 0; x2 < n; x2++) {
                    if (y1 == y2 && x1 == x2) continue;
                    long long s2 = 0;

                    if (y1 != y2)
                        s2 += riadky[y2] - d[y2][x2] - d[y2][x1];

                    if (x1 != x2)
                        s2 += stlpce[x2] - d[y2][x2] - d[y1][x2];

                    best = max(best, s1 + s2);
                }
            }
        }
    }

    cout << best << endl;
}
```

Vzorové riešenie

Na plný počet bodov musíme riešenie hrubou silou o jeden rád zrýchliť na $O(n^3)$, napríklad takto:

Najprv predpokladáme, že vysielače sú v optimálnom riešení v rôznych riadkoch a stĺpcoch. Vyskúšame každú možnú dvojicu stĺpcov: V oboch stĺpcoch si spočítame pokrytie jedného vysielača na každej pozícii v tomto stĺpci

(ignorujúc mrakodrap v druhom stĺpci, keďže ten bude určite pokrytý druhým vysielačom). Použijeme pritom predpočítané súčty riadkov a stĺpcov, podobne ako v riešení vyššie, aby sme každé pokrytie spočítali v $O(1)$. Ak tieto pokrytia zoradíme, môžeme jednoducho zobrať najlepšie pozície z ľavého a z pravého stĺpca (ak sú v tom istom riadku, skontrolujeme najlepšie dve pozície). Ich súčet nám potom dá najlepšie riešenie s vysielačmi v týchto dvoch stĺpcoch. Takto dosiehneme časovú zložitosť $O(n^3)$, keďže pre každú z $O(n^2)$ dvojíc stĺpcov raz prejdeme každý zo stĺpcov s n mrakodrapmi. Musíme si dať ale pozor, aby sme si namiesto zoraďovania iba vybrali dve najlepšie riešenia, keďže triedenie je príliš pomalé (triedenie $O(n \log n)$, celý cyklus teda $O(n^3 \log n)$).

Ešte musíme vyskúšať možnosť, že v optimálnom riešení sú oba vysielače v tom istom riadku alebo stĺpci: Stačí nám pre každý riadok vyskúšať všetky dvojice stĺpcov, v ktorých sa nachádzajú vysielače, a vybrať najlepšie pokrytie (a podobne pre stĺpce). Časová zložitosť $O(n^3)$ nám tu stačí, keďže takú má prvá časť tohoto riešenia.

Mimochodom, takáto možnosť naozaj môže nastať. Napríklad pre vstup nižšie sú v optimálnom riešení vysielače na mrakodrapoch s 0 *obyvateľmi:*

```
5
1 9 1 9 1
1 9 1 9 1
9 0 9 0 9
1 9 1 9 1
1 9 1 9 1
```

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0); cout.tie(0);

    int n; cin >> n;

    vector<vector<int>> d(n, vector<int>(n));
    vector<long long> riadky(n), stlpce(n);
    for (int y = 0; y < n; y++) {
        for (int x = 0; x < n; x++) {
            cin >> d[y][x];
            riadky[y] += d[y][x];
            stlpce[x] += d[y][x];
        }
    }

    long long best = 0;

    // rozne riadky
    vector<pair<long long, int>> stlpce1(n), stlpce2(n);
    for (int y1 = 1; y1 < n; y1++) {
        for (int y2 = 0; y2 < y1; y2++) {
            for (int x = 0; x < n; x++) {
                long long r1 = riadky[y1] - d[y1][x];
                long long r2 = riadky[y2] - d[y2][x];
                long long s = stlpce[x] - d[y1][x] - d[y2][x];
```

```
                // rovnaky riadok
                best = max(best, s + r1 + r2);

                // rozne riadky
                stlpce1[x] = make_pair(s + r1, x);
                stlpce2[x] = make_pair(s + r2, x);
            }

            // najdi 2 najvacsie
            partial_sort(stlpce1.begin(), stlpce1.begin()+2, stlpce1.end(), std::greater<pair<long long, int>>());
            partial_sort(stlpce2.begin(), stlpce2.begin()+2, stlpce2.end(), std::greater<pair<long long, int>>());
            sort(stlpce1.begin(), stlpce1.begin()+2, std::greater<pair<long long, int>>());
            sort(stlpce2.begin(), stlpce2.begin()+2, std::greater<pair<long long, int>>());

            if (stlpce1[0].second != stlpce2[0].second) {
                best = max(best, stlpce1[0].first + stlpce2[0].first);
            } else {
                best = max(best, stlpce1[0].first + stlpce2[1].first);
                best = max(best, stlpce1[1].first + stlpce2[0].first);
            }
        }
    }

    // rovnaky riadok
    for (int y = 0; y < n; y++) {
        for (int x1 = 1; x1 < n; x1++) {
            for (int x2 = 0; x2 < x1; x2++) {
                long long s1 = stlpce[x1] - d[y][x1];
                long long s2 = stlpce[x2] - d[y][x2];
                long long r = riadky[y] - d[y][x1] - d[y][x2];

                // rovnaky riadok
                best = max(best, r + s1 + s2);
            }
        }
    }

    cout << best << endl;
}
```

paulinia

8. Astrálne Kamene

(max. 12 b za popis, 8 b za program)

Málo kameňov

Ak je kameňov málo, vieme úlohu previesť na nasledovný podproblém:

Ak Jožko zoberie i-tý kameň, koľko najviac bodov vie odvtedy získať?

Riešenie je dynamické programovanie: Utrieďme si kamene podľa času dopadu a spracujme ich od najneskoršieho po najskorší. Pre každý už spracovaný kameň si zapamätáme hodnotu best_i - koľko najviac bodov vie

získať Jožko od momentu keď zoberie tento kameň. Túto hodnotu pre i-tý kameň spočítame nasledovne: pozrime sa na všetky kamene ktoré dopadli neskôr (pre ne sme už hodnotu spočítali). Takto zistíme ktoré má Jožko šancu zobrať a hodnota best_i je súčet hodnoty i-teho kameňa a maxima hodnôt pre dosiahnuteľné kamene.

Napokon si už len treba vybrať ktorým kameňom začneme. Na to stačí prejsť všetky kamene a odpoveďou je maximum z best_i.

Na zrekonštruovanie sekvencie si pre každý kameň stačí pamätať ktorý ďalší Jožko vezme.

Toto riešenie má časovú zložitosť $O(n^2)$, pamäťovú $O(n)$ a stačí na získanie 2 bodov z prvej sady.

Listing programu (C++)

```
#include<bits/stdc++.h>

using namespace std;

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

    vector<int> X(n), T(n), V(n);
    vector<pair<pair<int, int>, int> > coors;

    for(int i = 0; i < n; i ++) cin >> T[i];
    for(int i = 0; i < n; i ++) {
        cin >> X[i];
        X[i] -= p;
    }
    for(int i = 0; i < n; i ++) cin >> V[i];

    for(int i = 0; i < n; i ++) {
        int a2 = X[i] + T[i];
        int b2 = T[i] - X[i];

        if (a2 >= 0 && b2 >= 0) {
            coors.push_back({{T[i], X[i]}, i});
        }
    }

    sort(coors.begin(), coors.end());
    int m = coors.size();
    vector<long long int> best(m, 0);
    vector<int> to_use(m, 0);

    for(int i = 0; i < m; i ++) {
        long long int prior = 0;
        int from = -1;
        for(int j = 0; j < i; j ++) {
            if (coors[i].first.first - coors[j].first.first >= abs(coors[i].first.second - coors[j].first.second)) {
                if (prior < best[j]) {
                    prior = best[j];
                    from = j;
                }
            }
        }
        best[i] = prior + V[coors[i].second];
        to_use[i] = from;
    }

    long long int maxi = 0;
    int from = -1;

    for(int i = 0; i < m; i ++) {
        if (best[i] > maxi) {
            maxi = best[i];
            from = i;
        }
    }

    vector<int> which;
    while (from >= 0) {
        which.push_back(coors[from].second);
        from = to_use[from];
    }

    cout << which.size() << " " << maxi << endl;
    for(int i = 0; i < (int)which.size(); i ++) cout << (i ? " " : "") << which[i];
    cout << endl;

}
```

Veľa kameňov, v malom časopriestore

Čo keď je kameňov veľa, ale sú roztrúsené po malom časopriestore? V druhej sade sa kamene môžu vyskytovať iba prvých 5000 metrov a sekúnd. To je pomerne málo a vieme na tom založiť druhé riešenie dynamickým programovaním:

Pozrime sa na podproblém: Ak Jožko stojí v čase t na súradnici x, aký najväčší súčet vie odvtedy (vrátane) dosiahnuť?

Spracujme hodnoty od neskorších časov ku skorším. Všimnime si, že na keď je Jožko na nejakej časopriestorovej súradnici, sú tri možnosti kde skončí o sekundu neskôr: buď sa posunie o jedno miesto doľava $(x-1)$, o jedno doprava $(x+1)$, alebo ostane na mieste. Pre každú súradnicu sa tak stačí pozrieť na tri ďalšie a z nich vziať maximum (a prípadne pričítať hodnotu kameňa, ak tam, kde stojíme, nejaký je).

Riešenie - maximálny súčet hodnôt ktoré vie pozbierať, je hodnota pre súradnicu $t = 0$, $x = p$.

Rovnako ako v predchádzajúcom riešení, kamene ktoré Jožko vezme zrekonštruujeme tak, že že si pre každú súradnicu zapamätáme kde viedol optimálny krok.

Pre spočítanie zložitosti, označme X maximum z priestorových súradníc a T maximum časových súradníc. Časová aj pamäťová zložitosť je $O(XT)$ a toto riešenie stačí pre ďalšie dva body z druhej sady.

Listing programu (C++)

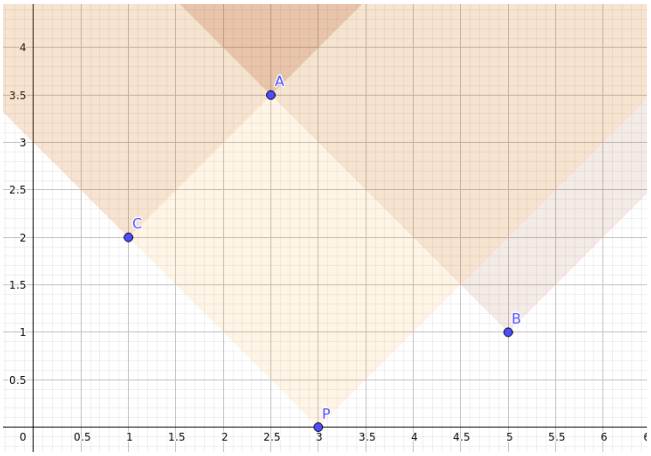
```
#include<bits/stdc++.h>

using namespace std;

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

    vector<int> X(n), T(n), V(n);
    vector<pair<pair<int, int>, int> > coors;

    for(int i = 0; i < n; i ++) cin >> T[i];
    for(int i = 0; i < n; i ++) {
        cin >> X[i];
        X[i] -= p;
    }
    for(int i = 0; i < n; i ++) cin >> V[i];

    for(int i = 0; i < n; i ++) {
        int a2 = X[i] + T[i];
        int b2 = T[i] - X[i];

        if (a2 >= 0 && b2 >= 0) {
            coors.push_back({{T[i], X[i]}, i});
        }
    }

    sort(coors.begin(), coors.end());
    int m = coors.size();
    vector<long long int> best(m, 0);
    vector<int> to_use(m, 0);

    for(int i = 0; i < m; i ++) {
        long long int prior = 0;
        int from = -1;
        for(int j = 0; j < i; j ++) {
            if (coors[i].first.first - coors[j].first.first >= abs(coors[i].first.second - coors[j].first.second)) {
                if (prior < best[j]) {
                    prior = best[j];
                    from = j;
                }
            }
        }
        best[i] = prior + V[coors[i].second];
        to_use[i] = from;
    }

    long long int maxi = 0;
    int from = -1;

    for(int i = 0; i < m; i ++) {
        if (best[i] > maxi) {
            maxi = best[i];
            from = i;
        }
    }

    vector<int> which;
    while (from >= 0) {
        which.push_back(coors[from].second);
        from = to_use[from];
    }

    cout << which.size() << " " << maxi << endl;
    for(int i = 0; i < (int)which.size(); i ++) cout << (i ? " " : "") << which[i];
    cout << endl;

}
```

Vzorové riešenie

Vzorové riešenie má myšlienku podobnú prvému riešenie hrubou silou - pre každý kameň si položíme otázku: aký najväčší súčet hodnôt môže Jožko získať od momentu ako zoberie i-tý kameň?

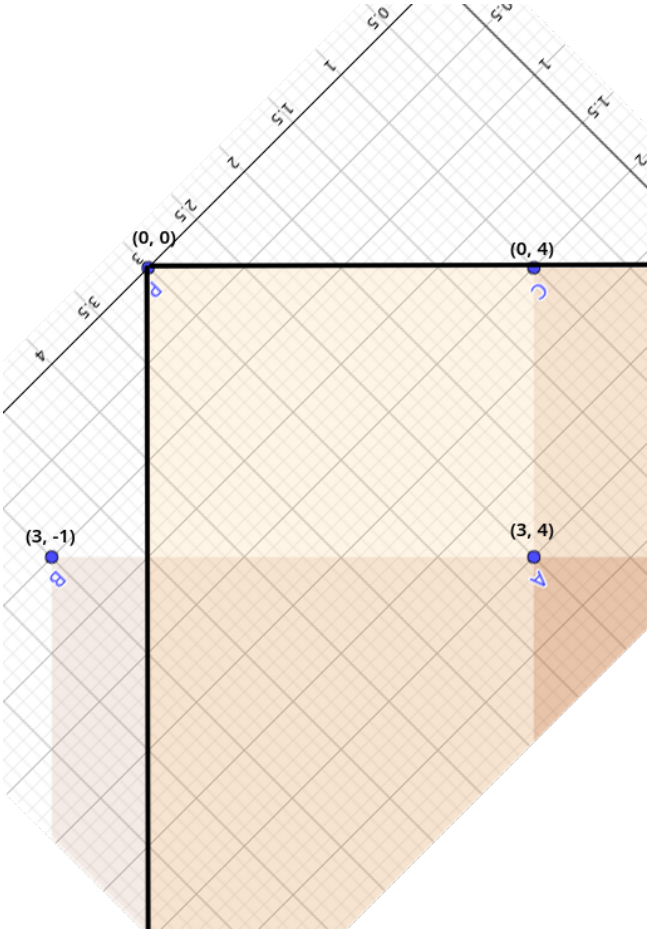
V riešení na prvú sadu sme si všetkých kandidátov po jednom prezreli, a vybrali maximum. Vyzerá to tak, že by to chcelo nejaký spôsob, ako rýchlo zistiť maximum z vhodnej množiny kameňov.

Predstavme si situáciu geometricky v časopriestore a zaznačme si z ktorých kameňov sa dá dostať na ktoré.



Všimnime si, že jednoduché pravidlo ako "ako ďalší kameň môžme vziať akýkoľvek, čo padne neskôr" nefunguje, napríklad na obrázku kameň B zo štartovnej pozície nie je dosiahnuteľný. Ak nájdeme pekné súradnice, v ktorom kritérium "viem zobrať tento kameň začínajúc na pozícii (x, t)" sa dá vyjadriť ako interval problém by sa výrazne zjednodušil.

Na druhý pohľad si vieme všimnúť, že výsek časopriestoru do ktorého sa vieme od nejakého kameňu dostať je pravouhlý a ak kameň 1 leží vo výseku kameňa 2 a kameň 2 leží vo výseku kameni 3, tak kameň 1 leží vo výseku kameňa 3. Skúsme obrátiť súradnice o 45 stupňov:



Môžme si všimnúť, že premenením súradníc na (a, b)

$$a = \frac{(x - p) + t}{2}$$

$$b = \frac{t - (x - p)}{2}$$

Aby sme sa vyhli desatinným číslam, môžme vynechať delenie dvomi -

Dostaneme súradnice, v ktorých sa z (a, b) vieme dostať na všetky body (s, t) kde $s \leq a$ a $t \leq b$.

Teraz vieme úlohu riešiť pomocou intervalového stromu: spracujme kamene v poradí najskôr od najväčšieho a a potom od najväčšieho b. Pri spracovaní kameňa na pozícii (a, b) sa pozrieme do intervalového stromu na pozície od b väčšie a v logaritmickom čase zistíme ktorý kameň je optimálne vziať a koľko tak získame (potrebujeme obe tieto informácie). Následne uložíme informáciu o kameni na b-tú pozíciu do intervaláča.

Akú časovú zložitosť má toto riešenie? b-súradnica, ktorou indexujeme v intervaláči je ohraničená T-najväčším vyskytujúcim sa časom. Vytvoriť intervalový strom má lineárnu časovú i pamäťovú zložitosť, následné spracovanie každého kameňa $\log T$ čas. Celková časová zložitosť je tým pádom $O(T + n \log T)$. Toto riešenie by vyriešilo druhú a tretiu sadu a získalo by tak 4 body.

Pozorný čitateľ sa už isto čuduje? Nehovoril posledný nadpis vzorové riešenie? Pravdou je, že sme pri ňom už veľmi blízko - problém je veľkosť intervaláča - možný rozsah súradníc je rádovo viac, ako počet kameňov. Valná väčšina pozícií v intervalovom strome ostáva nevyužitá, ale spôsobuje TLEtie programu.

Riešenie na tento problém sa volá *kompresia súradníc*. Namiesto používanie b na indexovanie, prečíslujme súradnice ešte raz: zaujíma nás iba relatívne poradie (ktoré kamene majú nižšiu/vyššiu b súradnicu), takže nová druhá súradnica číslo "koľko z všetkých kameňov má nižšiu b-súradnicu ako ja?". Toto číslo sa dá zistiť utriedením podľa b-súradnice v $O(n \log n)$ čase.

Prečíslovanie spôsobí, že intervaláč bude mať veľkosť $O(n)$ a teda sa časová zložitosť upraví na $O(n \log n)$, pamäťová na $O(n)$ a riešenie prejde na všetkých sadách.

Listing programu (C++)

```
#include<bits/stdc++.h>

using namespace std;

struct intervalac {
    int N;
    vector<int> start, range;
    vector<pair<long long int, long long int>> maxi;

    intervalac(int n) {
        N = pow(2, ceil(log2(max(n, 1))));

        start.resize(2 * N, 0);
        range.resize(2 * N, N);
        maxi.resize(2 * N, {0, -1});

        for (int i = 2; i < 2 * N; i ++) {
            range[i] = range[i / 2] / 2;
            start[i] = start[i / 2] + (i % 2) * range[i];
        }
    }

    pair<long long int, long long int> get_maxi(int from, int id) {
        if (start[id] + range[id] <= from) return {0, -1};
        if (start[id] >= from) return maxi[id];
        return max(get_maxi(from, id * 2), get_maxi(from, id * 2 + 1));
    }

    void update(int p, long long int val, int index, int id) {
        if (start[id] > p || start[id] + range[id] <= p) return;
        if (range[id] == 1) {
            maxi[id] = {val, index};
            return;
        }
        update(p, val, index, id * 2); update(p, val, index, id * 2 + 1);
        maxi[id] = max(maxi[id * 2], maxi[id * 2 + 1]);
    }
};

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

    vector<int> X(n), T(n), V(n), to_use(n, -1);
    vector<pair<pair<int, int>, int> > coors;

    for (int i = 0; i < n; i ++) cin >> T[i];
    for (int i = 0; i < n; i ++) {
        cin >> X[i];
        X[i] -= p;
    }
    for (int i = 0; i < n; i ++) cin >> V[i];

    set<pair<int, int>> compress;
    map<pair<int, int>, int> translate;

    for(int i = 0; i < n; i ++) {
        int a2 = X[i] + T[i];
        int b2 = T[i] - X[i];

        if (a2 >= 0 && b2 >= 0) {
            coors.push_back({{a2, b2}, i});
```

```
            compress.insert({b2, a2});
        }
    }

    int l = 0;
    for (auto c : compress) {
        translate[c] = l;
        l ++;
    }

    intervalac I(l);

    sort(coors.rbegin(), coors.rend());

    int m = coors.size();

    for (auto vec : coors) {
        int i = translate[{vec.first.second, vec.first.second}];
        pair<long long int, long long int> max_val = I.get_maxi(i, 1);
        I.update(i, max_val.first + V[vec.second], vec.second, 1);
        to_use[vec.second] = max_val.second;
    }

    pair<long long int, long long int> maxi = I.get_maxi(0, 1);
    vector<int> which;
    int take = maxi.second;

    while (take >= 0) {
        which.push_back(take);
        take = to_use[take];
    }

    cout << which.size() << "␣" << maxi.first << endl;
    for(int i = 0; i < which.size(); i++) cout << (i ? "␣" : "") << which[i];
    cout << endl;
}
```