PRASK
VII. ročník, 2020/21
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola letnej časti

1. Permoníci

vzorák napísal(a) Roman
(max. 100 b za riešenie)

K tejto úlohe sme pripravili videovzorák[1].

2. Rámovanie

vzorák napísal(a) Adam
(max. 0 b za riešenie)

Táto úloha bola náročná skôr jej implementáciou ako znalosťou algoritmov a potrebných postupov. Bolo počas nej nutné spracovať a upravovať väčšie množstvo textových dát.

Teória

Pri riešení je dôležité si uvedomiť aké informácie potrebujeme na zarámovanie jednej mozaiky. Keďže zo zadania vieme, že mozaika bude vždy len jedna stačí nám zistiť jej výšku, šírku a polohu v priestore - potom budeme schopný okolo nej vytvoriť rám. Všetky tieto údaje vieme nahradiť aj štyrmi najvzdialenejšími bodmi (hranicami) od jej stredu, ktoré budú teda reprezentovať miesta, kde vykreslíme 4 steny obdĺžnikového rámu. Nájsť sa dajú rôzne, ale v najhoršom prípade, budú mať riešenia vždy časovú zložitosť $O(n^2)$. Čo vôbec nieje problém, keďže už len vykresľovanie výstupu z pamäte bude mať rovnakú časovú zložitosť.

Tieto štyri hranice mozaiky vieme nájsť postupne zametaním[2] z každej strany, alebo počas jednej iterácie za stáleho sledovania znakov patriacich mozaike. Čo vlasten znamená, že pri každom znaku sa pozriem, či náhodou nieje zatiaľ najvzdialenejší od stredu, resp. najbližší k jednej zo stien plátna. Ak to je takýto bod, tak si ho zapmätám a ďalej hľadám body, ktoré sú lepšie ako tento. Následne pri vykresľovaní výsledného obrázku, budeme dbať na medze vyhradené hranicami a namiesto bodiek dosadíme na správne miesta znaky #.

Riešenie

Riešenie v c++ mohlo vyzerať napríklad takto:

Listing programu (C++)

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

int main() {
    int width, height;
    cin >> height >> width;
    vector<vector<char>> obrazok;

    // HORE, DOLE, VPRAVO, ĽVAVO
    int limity[4] = {-1, -1, -1, -1};

    for (int i=0; i < height; i++) {
        obrazok.push_back({});
        for (int j=0; j < width; j++) {
            char a;
            cin >> a;
            if (a != '.') {
```

[1] https://www.youtube.com/watch?v=LB2KaYRYRMY
[2] "Zametanie" sa nazýva technika počas ktorej postupne prechádzam z jednej strany na druhú a zaznamenámvam si počet výskytov hľadaných znakov, alebo čísel.

```
                if (limity[0] > i || limity[0] == -1) limity[0] = i;
                if (limity[1] < i || limity[1] == -1) limity[1] = i;
                if (limity[2] < j || limity[2] == -1) limity[2] = j;
                if (limity[3] > j || limity[3] == -1) limity[3] = j;
            }
            obrazok[i].push_back(a);
        }
    }

    limity[0]--;
    limity[1]++;
    limity[2]++;
    limity[3]--;

    if (limity[0] >= 0 && limity[3] >= 0 &&
        limity[1] < height && limity[2] < width) {
        for (int i=limity[0]; i <= limity[1]; i++) {
            for (int j=limity[3]; j <= limity[2]; j++) {
                if (limity[1] > i && i > limity[0] && limity[2] > j && j >
                    ↪ limity[3]) continue;
                obrazok[i][j] = '#';
            }
        }

        for (auto i : obrazok) {
            for (auto j : i) {
                cout << j;
            }
            cout << endl;
        }
    } else {
        cout << "Neda sa" << endl;
    }

    return 0;
}
```

Riešenie v pythone mohlo vyzerať napríklad takto:

Listing programu (Python)

```
height, width = map(int, input().split())

limity = [-1, -1, -1, -1]
obrazok = []

for i in range(height):
    obrazok.append(list(input()))
    for j in range(width):
        if (obrazok[i][j] != '.'):
            if (limity[0] > i or limity[0] == -1): limity[0] = i
            if (limity[1] < i or limity[1] == -1): limity[1] = i
            if (limity[2] < j or limity[2] == -1): limity[2] = j
            if (limity[3] > j or limity[3] == -1): limity[3] = j

limity[0]-=1
limity[1]+=1
limity[2]+=1
limity[3]-=1
```

```
if (limity[0] >= 0 and limity[3] >= 0 and
        limity[1] < height and limity[2] < width):
    for i in range(limity[0], limity[1]+1, 1):
        for j in range(limity[3], limity[2]+1, 1):
            if (limity[1] > i and i > limity[0] and limity[2] > j and j > limity
                ↪ [3]): continue
            obrazok[i][j] = '#'
    for i in obrazok:
        for j in i:
            print(j, end="")
        print()
else:
    print("Neda sa")
```

vzorák napísal(a) MisQo
(max. 0 b za riešenie)

3. Aktuálny problém

V tejto úlohe nám stačí spraviť obyčajnú maximovú haldu, ktorá bude zvládať dve operácie - push a pop. To ako, prečo, ako rýchlo... halda funguje bolo v štúdijnom texte o Halde[3]. Teraz si ukážeme už len implementáciu.

Python:

Listing programu (Python)

```
pole = []


def push(co):
    pole.append(co)
    kde = len(pole) - 1
    while kde > 0:
        rodic = (kde - 1) // 2
        if pole[kde] > pole[rodic]:
            pole[kde], pole[rodic] = pole[rodic], pole[kde]
            kde = rodic
        else:
            break


def pop():
    n = len(pole)
    pole[0], pole[n-1] = pole[n-1], pole[0]
    pole.pop()
    kde = 0
    while True:
        kam = kde
        if 2*kde+1 < n-1 and pole[2*kde+1] > pole[kam]:
            kam = 2*kde+1
        if 2*kde+2 < n-1 and pole[2*kde+2] > pole[kam]:
            kam = 2*kde+2
        if kam != kde:
            pole[kde], pole[kam] = pole[kam], pole[kde]
            kde = kam
        else:
            break
```

[3] https://prask.ksp.sk/strudlohy/halda/

```
n = int(input())

for i in range(n):
    t, x = map(int, input().split())
    if t == 0:
        pop()
        if len(pole) == 0:
            print('prazdna')
        else:
            print(pole[0])
    if t == 1:
        push(x)
        print(pole[0])
```

C++:

Listing programu (C++)

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

using namespace std;

void push(int co, vector<int> &p)
{
    p.push_back(co);
    int kde = p.size() - 1;
    while (kde > 0)
    {
        int rodic = (kde-1)/2;
        if (p[kde] > p[rodic])
        {
            swap( p[kde], p[rodic] );
            kde = rodic;
        } else break;
    }
}

void pop(vector<int> &p)
{
    int n = p.size();
    swap( p[0], p[n-1] );
    p.pop_back();
    int kde = 0;
    while (true)
    {
        int kam = kde;
        if (2*kde+1 < n-1 && p[2*kde+1] > p[kam]) kam = 2*kde+1;
        if (2*kde+2 < n-1 && p[2*kde+2] > p[kam]) kam = 2*kde+2;
        if (kam != kde)
        {
            swap( p[kde], p[kam] );
            kde = kam;
        }
        else break;
    }
}
```

```
int main()
{

    vector<int> pole;

    int n;
    cin >> n;

    for(int i = 0; i < n; i++)
    {
        int t, x;
        cin >> t >> x;
        if(t == 0)
        {
            pop(pole);
            if(pole.size() == 0)
            {
                cout << "prazdna\n";
            }
            else
            {
                cout << pole[0] << '\n';
            }
        }

        if(t == 1)
        {
            push(x, pole);
            cout << pole[0] << '\n';
        }
    }
}
```

vzorák napísal(a) MisQo
(max. 0 b za riešenie)

4. Stovky programátorov

V tejto úlohe potrebujeme udrživiať čísla tak trochu usporiadané. To sa dá robiť pomocou rôznych pomerne zložitých dátových štruktúr. My si ukážeme riešenie, ktoré využíva iba haldu. To ako, prečo, ako rýchlo… halda funguje bolo v študíjnom texte o Halde[4].

Riešenie s jednou haldou

Vieme že táto úloha má byť o halde. Skúsime sa teda zamyslieť ako by sme vedeli haldu využiť. Vlastnosť haldy je, že vieme jednoducho nájsť ktorý prvok je najväčší (prípadne najmenší). O ostatných prvkov však nevieme skoro nič povedať. Jedno z riešení, by teda mohlo byť že si budeme skúseností programátorov držať v halde. Potom v každom kroku z našej haldy vyberieme polovicu prvkov, zistíme hodnotu mediánu, a následne všetky prvky naspať vrátime. Takéto riešenie je však veľmi pomalé.

Optimalizácia predošlého riešenia

Môžeme si všimnúť, že v predošlom riešení robíme veľa "roboty" zbytočne. Na konci každého kroku najprv polovicu prkov vrátime len na to, aby sme ich neskôr skoro všetky znovu z haldy vybrali. Toto je samozrejme zbytočné. Nechajme teda polovicu prvkov stále vybranú.

Máme teda prvky rozdelené na dve rovnako veľké skupiny, v jednej sú prvky väčšie ako medián, v druhej prvky menšie alebo rovné ako medián. Teraz vieme jednoducho vypísať hodnotu mediánu, je to totiž najväčší prvok z druhej skupiny. Ako si však vieme udržiavať tieto skupiny, keď budú pribúdať nový programátori?

Keď nám príde nové číslo(nový programátor) najprv zistíme do ktorej skupiny by malo patriť. Tak ho tam teda pridáme. Teraz by mohol nastať problém, a to, že v tejto skupine je teraz veľa prvkov. Vtedy treba

[4] https://prask.ksp.sk/strudlohy/halda/

jeden prvok presunúť do druhej skupiny (môžete si premyslieť, prečo určite stačí presunúť jeden prvok). Prvok, ktorý musíme presunúť je ten najväčší/najmenší z danej skupiny, aby stále platilo, že skupiny sú rozdelené podľa veľkosti. Keďže z času na čas potrebujeme zistiť, a vybrať najväčší resp. najmenší prvok, budeme si naše skupiny ukladať ako haldy.

Zhrnutie

Budeme mať teda dve haldy, jednu maximovú a jednu minimovú. V maximovej bude polovica(zaokrúhlené nahor) všetkých prvkov, a budú to tie menšie. V minimovej zase budú tie väčšie prvky. Vždy keď dá svoju ponuku nový programátor najprv ho pridáme a upravíme skupiny, a potom jednoducho nájdeme vhodného programátora.

Časová zložitosť

V každom kroku, teda vždy keď pridávame nejakého programátora spravíme niekoľko operácií s haldami. Každá táto operácia nám zaberie najviac $O(logn)$, a teda keďže týchto krokov spravíme n, celková zložitosť bude $O(nlogn)$.

Implementácia

V kóde využijeme to, že haldu niekto pred nami už naprogramoval. V c++ využijeme priority_queue.

Listing programu (C++)

```
#include <iostream>
#include <queue>

using namespace std;

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

    priority_queue<int, vector<int>, greater<int>> minimova;
    priority_queue<int, vector<int>, less<int>> maximova;

    for(int i = 0; i < n; i++)
    {
        int x;
        cin >> x;

        if(maximova.size() == 0 || x <= maximova.top())
        {
            maximova.push(x);
        }
        else minimova.push(x);

        if(maximova.size() < minimova.size())
        {
            maximova.push(minimova.top());
            minimova.pop();
        }
        else if(maximova.size() >= minimova.size() + 2)
        {
            minimova.push(maximova.top());
            maximova.pop();
        }

        cout << maximova.top() << endl;
    }
```

```
    return 0;
}
```

5. Krádež rezňov

0 bodov za riešenie

Prvý krok, ktorý musíme spraviť k úspešnému naprogramovaniu úlohy, je zistiť víťaznú stratégiu. Následné naprogramovanie je potom relatívne priamočiare.

Stratégia

Na ukážku si zoberieme príklad zo zadania. Na kôpke je 17 rezňov a môžeme si zobrať jeden až päť rezňov. Najprv si položíme otázku: ak by na kôpke bol práve jeden rezeň, kto by vyhral? Odpoveď je jednoduchá: ten/á hráč/ka ktorý/á je na ťahu. Tak isto vyhrá hráč/ka na ťahu, ak sú na kôpke dva rezne, môže si ich proste zobrať. Rovnako to je aj pre 3, 4 a 5 rezňov. Čo sa ale stane keď je na kôpke 6 rezňov?

Keď si v tomto prípade hráč na ťahu zoberie hocikoľko rezňov, môže nastať 5 situácií. Ak si zoberie 1, ostane na kôpke 5, ak zoberie 2, ostanú 4 atď. Ak zoberie 5, na kôpke ostane jeden. Ako sme pred chvíľou ukázali, všetky z týchto možností vyhrá hráč na ťahu. Vieme teda povedať, že pozícia so 6 rezňami je prehrávajúca. Ak sa nám podarí dostať protivníka do prehrávajúcej pozície, vieme s istotou povedať, že vieme vyhrať. Taktiež vieme povedať, že pozície, z ktorých vieme dostať protivníka do prehrávajúcej pozície, sú vyhrávajúce. V tomto prípade sú to čísla 7, 8, 9, 10 a 11. Keď zo 7 zoberieme 1, z 8 zoberieme 2 atď, dostaneme protivníka na 6, čo je prehrávajúca pozícia. 12 následne bude znova prehrávajúce, pretože sa z toho dá dostať iba do vyhrávajúcej pozície. Môžeme si tu všimnúť isté pravidlo: vyhrávajúce a prehrávajúce zóny sa striedajú.

Zovšeobecnenie

Z kôpky si vždy vieme zobrať najmenej `n` a najviac `m-1` rezňov. To znamená, že ak je na kôpke menej ako `m-1` rezňov, vieme si zobrať všetky a vyhrať. Nasleduje prehrávajúca zóna. Prehrávajúca je preto, že sa z nej dá dostať iba do vyhrávajúcej. Minimálne bude mať dĺžku 1, pretože predchádzajúcich čísel je vyhrávajúcich. Ak je ale `n` väčšie ako 1, bude aj prehrávajúca zóna väčšia, pretože sa z nasledujúcich čísel nedá dostať do menších čísel tej istej prehrávajúcej zóny.

Z tohoto si vieme odvodiť, že tieto vyhrávajúce a prehrávajúce zóny sa striedajú, veľkosť vyhrávajúcej zóny je `m-1` a veľkosť prehrávajúcej zóny je `n`.

Optimalizácie

Je lákavé si ručne odrátať všetky výherné a prehrávajúce zóny. Avšak pri veľkom množstve rezňov a malom m bude náš program veľmi pomalý. Toto sa však dá vyriešiť viacerými spôsobmi. Aktuálna kopa rezňov sa dá zmodulovať `(n+m-1)`, pretože to reprezentuje zvyšok zóny po odstránení dvojíc výherná + prehrávajúca. Tento spôsob je použitý aj vo vzorovom kóde.

Ďalší prístup by bol predrátať si pre každé číslo či je vyhrávajúce alebo prehrávajúce, a následne už len vracať toto z poľa podľa toho ako hrá oponent. Toto nám však zvýši pamäťovú zložitosť, keďže si musíme pamätať každý ťah.

Vzorové riešenie

```
def tahaj(k, n, m):
    velkost_vyhravajucej_zony = m - 1
    velkost_prehravajucej_zony = n

    # Optimalizácia používajúca modulo
    i = k - (k % (velkost_vyhravajucej_zony + velkost_prehravajucej_zony))
    vyhravam = False
    while i < k:
        if not vyhravam:
            i += velkost_vyhravajucej_zony
        else:
            i += velkost_prehravajucej_zony
        vyhravam = not vyhravam
```

```
    if not vyhravam:  # Nikdy by sme sa nemali dostať do prehrávajúcej zóny
        raise ValueError
    else:
        return max(n, k % (velkost_vyhravajucej_zony + velkost_prehravajucej_zony))


reznov, najmenej, najviac = list(map(int, input().split()))

while True:
    tah = tahaj(reznov, najmenej, najviac)
    print(tah)
    reznov -= tah
    inp = input()
    try:
        reznov -= int(inp)
    except ValueError:
        break
```