Arhiva educațională

2023

Atenție: Conținutul acestui PDF poate fi învechit.
Cea mai recentă versiune se află la
https://github.com/stefdasca/RoAlgo-Educational-Archive.

Cuprins

Partea I.

Începători

1. Căutare binară

1.1. Problema inițială

Să presupunem că avem un șir de N numere și memorie astfel încât să putem reține *doar* șirul (plus evident alte variabile, dar nu foarte multe). Noi primim mai multe întrebări, de forma: Există valoarea X în șir?

În mod evident, o soluție foarte trivială este să parcurgem manual șirul pentru fiecare întrebare, și să vedem dacă elementul cerut apare sau nu în șir, astfel obținând complexitate totală de $O(N \cdot Q)$. Singura noastră problemă este că noi o să avem N și Q undeva în jur de 10^6, ceea ce va face ca această abordare să pice clar în timp, deci va trebui găsită o soluție mult mai eficientă. Aici intervine algoritmul de *căutare binară*.

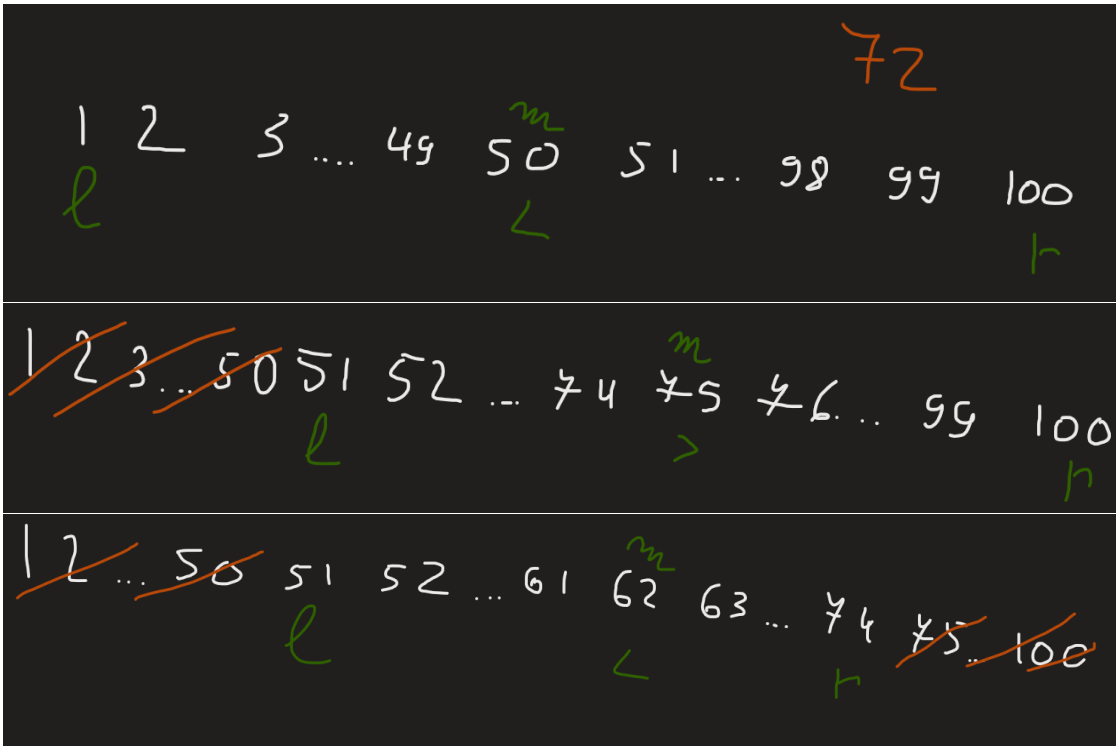
1.2. Prezentarea algoritmului

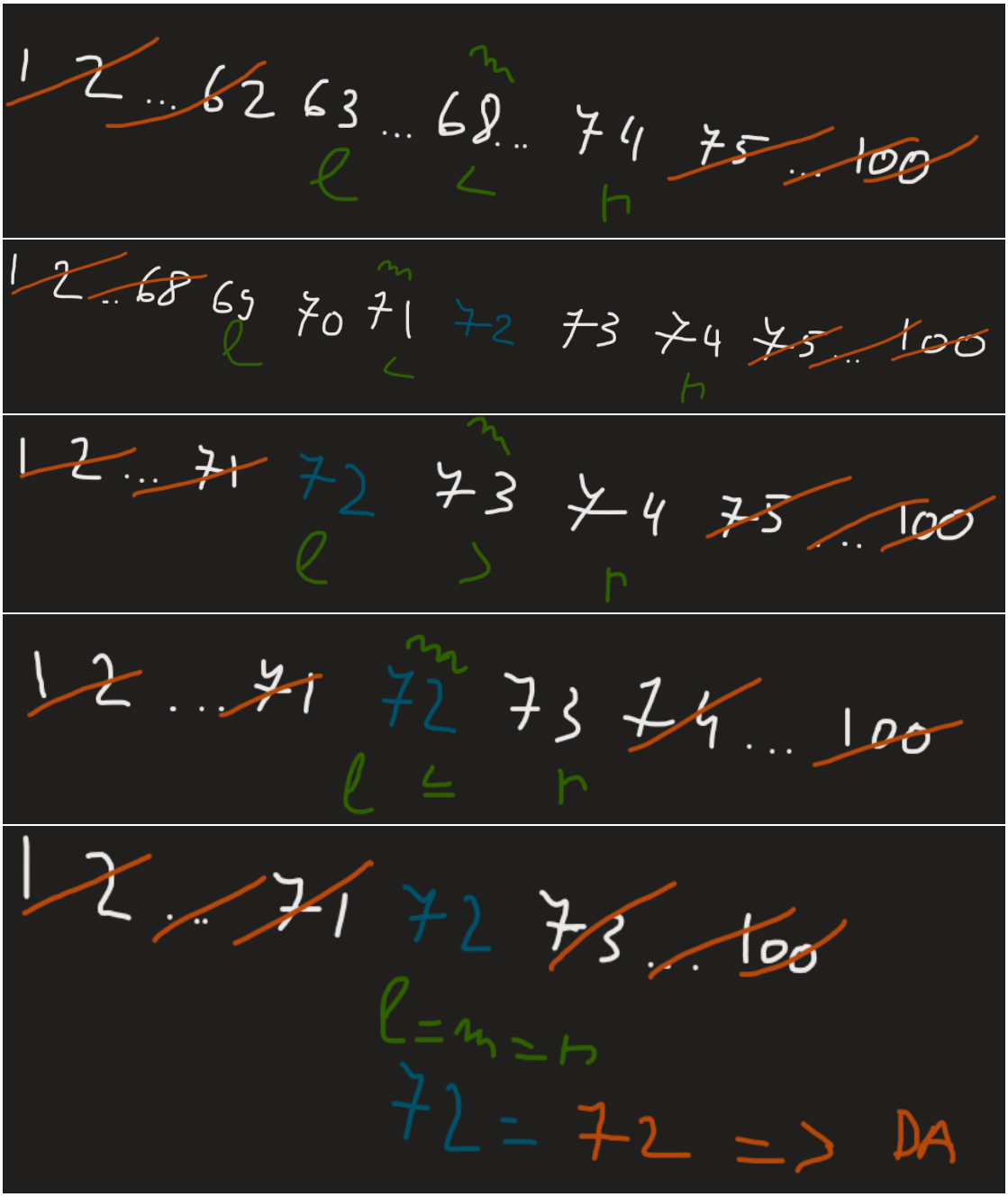
Notă: De acum încolo, se va presupune că șirul nostru este sortat crescător. Căutarea binară pe un șir nesortat va da mereu răspunsuri eronate.

În algoritmul de căutare binară se va pleca de la analiza șirului pe întreaga sa lungime, și se va fixa punctul de mijloc din sir. Dacă valoarea poziției din mijloc este mai mică decât valoarea căutată, atunci sigur valoarea căutată se poate (că nu știm sigur dacă există!) afla în a doua jumătate, altfel, se poate afla în prima jumătate. Mai departe, nu va mai fi necesar sa analizăm tot șirul,

ci doar jumătatea relevantă (cea în care considerăm noi că există o șansă să găsim valoarea noastră), și algoritmul se va repeta până când lungimea devine 1 și putem determina răspunsul. Dat fiind faptul că noi la fiecare pas împărțim la 2 lungimea șirului, acest lucru ne va da complexitate logaritmică la determinarea răspunsului, deci vom avea complexitate $O(Q \log N)$ (dacă șirul nostru nu este sortat din input, se mai adaugă și un $O(N \log N)$ la complexitate), cu memorie $O(N)$.

Pentru o înțelegere mai clară a algoritmului, să presupunem următorul exemplu: se dă un șir sortat crescător unde apar toate numerele de la 1 la 100, și se cere să determinăm dacă există în șir valoarea 72.





O întrebare la care trebuie totuși dat răspuns este: De ce împărțim în două jumătăți și de ce nu în 3 treimi? Da, $\log_3 N < \log_2 N$, dar numărul de verificări efectuate va fi mai mare la împărțirea în 3 treimi, deci în continuare este mai eficient să împărțim în două jumătăți. În mod inductiv se va demon-

stra pentru orice împărțire posibilă.

1.3. O implementare banală

Cea mai des întâlnită implementare a căutării binare este următoarea:

```
void cb1 (int n)
{
    int l = 1, r = n, ans = -1;
    while (l <= r)
    {
        int mij = (l + r) / 2;
        if (conditie)
        {
            ans = mij;
            l = mij + 1;
        }
        else
        {
            r = mij - 1;
        }
    }
}
```

Implementarea de mai sus este una corectă, dar se pot întâlni următoarele bug-uri:

- Schimbarea în $l = mij$ și $r = mij$ va face ca programul nostru să ruleze într-o buclă infinită (deoarece ambele valori vor atinge la un moment dat valoarea *mij*, și deci va fi respectată mereu condiția $l \leq r$)
- În timp ce-l calculăm pe *mij*, ne putem lua overflow (dacă prin absurd ajungem să căutam fix pe la valorile maxime pe care le poate reține tipul

nostru de date, este inevitabil un overflow generat de $l + r$). De aceea, următoarea variantă prezentată se va axa fix pe rezolvarea acestui bug.

1.4. O implementare corectă

```
void cb2 (int n)
{
    int l = 1, r = n, ans = -1;
    while (l < r)
    {
        int mij = l + (r - l) / 2;
        if (conditie)
        {
            ans = mij;
            l = mij + 1;
        }
        else
        {
            r = mij - 1;
        }
    }
}
```

Această căutare binară se bazează pe principiul menționat mai sus: noi înjumătățim de fiecare dată lungimea șirului pe care încercăm să căutăm ceea ce ne interesează. Formula de mai sus pentru calcularea mijlocului este echivalentă cu cea din prima căutare, dar mai mult, nu are cum să ne dea overflow.

De fiecare dată când mijlocul nostru verifică *condiție*, noi facem un *„salt”* de la o poziție l la alta. La finalul căutării, indicele l final va fi defapt o sumă a salturilor, iar ca pe orice număr întreg, noi acest număr îl putem descompune într-o altă bază numerică. Hai să vedem cum putem rafina această idee cu o

altă implementare mai jos.

1.5. Căutarea binară a lui Mihai Pătrașcu

```
void cb3_patrascu (int n)
{
    int l = 0, e = 31;
    while (e >= 0)
    {
        if (l + (1 << e) <= n && conditie)
        {
            l += (1 << e);
        }
        e--;
    }
}
```

Baza în care noi vom descompune suma va fi baza 2, pentru a menține în continuare complexitatea $\log_2 N$. Inițial, vom pleca cu un exponent e, unde 2^e va reprezenta lungimea secvenței pe care o analizăm (atenție să nu ieșim din vector!). Chiar dacă noi vom analiza inițial o lungime care este putere de 2, care foarte probabil să fie diferită de N, se poate demonstra foarte ușor că noi (dacă o să fie necesar), vom putea căuta valori și în acea secvență neacoperită inițial. Lăsăm această demonstrație ca temă pentru cititor.

Căutarea de mai sus poartă și numele de *Căutarea binară a lui Mihai Pătrașcu*, sau *căutarea pe biți*.

În mare parte, aceste căutări binare ne vor da aceeași complexitate peste tot, în schimb, când vrem să implementăm algoritmul de Lowest Common Ancestor (LCA) cu Binary Lifting, căutarea binară pe biți reduce algoritmul de la $O(\log^2 H)$ la $O(\log H)$, unde H reprezintă adâncimea maximă a arbore-

lui.

1.6. Concluzii și lecturi suplimentare

Căutarea binară este unul dintre cele mai fundamentale principii ale algoritmicii, fiind absolut necesar pentru a optimiza probleme unde ni se cere să determinăm existența unei valori într-un șir, sau determinarea unui număr maxim / minim care să respecte o condiție impusă de problemă etc.

Pentru aprofundarea a algoritmului, recomand rezolvarea următoarelor probleme și citirea următoarelor articole:

- Problema cautbin (Infoarena)

Partea II.

Mediu

2. Divizibilitate

De-a lungul parcursului vostru în domeniul algoritmicii, precum și de multe ori în diferite olimpiade și concursuri de informatică, va trebui să rezolvați multe probleme care se bazează pe un fundament matematic, studiul teoriei din spatele divizibilității numerelor naturale precum și a algoritmilor de aflare a numerelor prime, numărului de divizori, lucrului eficient cu numerele prime devenind toate foarte importante pentru asimilarea în cel mai bun mod posibil a acestui capitol. Totuși, acest document reprezintă doar un punct de plecare în ceea ce privește aplicațiile teoriei numerelor în algoritmică, alte concepte fiind discutate în documentele ulterioare. Aceste noțiuni se vor găsi foarte des în problemele de informatică pentru clasele de gimnaziu și clasa a IX-a.

2.1. Noțiuni introductive

Definiția 2.1.1. *Un număr x este numit **divizor** al altui număr y, dacă y se poate scrie ca produsul dintre x și un alt număr întreg t.*

Observație. Orice număr *n* se împarte la 1 și la el însuși.

Definiția 2.1.2. *Definim un **divizor comun** al unei perechi de numere* (*a, b*) *ca fiind un număr c care este un divizor atât al lui a, cât și al lui b.*

Definiția 2.1.3. *Definim **cel mai mare divizor comun** (CMMDC) al unei perechi de numere* (a, b) *ca fiind cel mai mare număr care este un divizor atât al lui a, cât și al lui b. Vom nota* $x = (a, b)$.

Definiția 2.1.4. *Definim **cel mai mic multiplu comun** (CMMMC) al unei perechi de numere* $[a, b]$ *ca fiind cel mai mic număr care este un multiplu atât al lui a, cât și al lui b. Vom nota* $x = [a, b]$.

Observație. $a \cdot b = (a, b) \cdot [a, b]$. Drept concluzie, $(a, b) = \frac{a \cdot b}{[a,b]}$.

Pentru aflarea celui mai mare divizor comun a două numere, există doi algoritmi principali. Primul dintre ei se bazează pe scăderi repetate, la fiecare pas scăzându-se din numărul mai mare, numărul mai mic până când cele două valori devin egale. Deși pentru multe perechi de numere acest algoritm este destul de eficient, atunci când diferența dintre numere este foarte mare, algoritmul va rula în timp cvasi-liniar (de exemplu, pentru numerele 3 și 10^9, un calculator are nevoie de câteva secunde să afle CMMDC-ul folosind acest algoritm).

De aceea vom folosi algoritmul lui Euclid prin împărțiri repetate pentru a ajunge la răspuns. Acest algoritm pleacă de la ideea că o slăbiciune majoră a algoritmului prin scăderi este dată de situația când raportul dintre numărul mai mare și cel mai mic este foarte mare, când practic efectuăm aceeași operație de foarte multe ori. De aceea, în loc de scăderi, la fiecare pas vom afla restul împărțirii numărului mai mare la cel mai mic, înlocuind posibilele operații de scădere cu o singură împărțire, algoritmul devenind mult mai eficient.

De exemplu, să analizăm numerele 40 și 18.

- $a = 40$, $b = 18$. $a \,\%\, b = 4$, noile valori fiind $a = 18$, $b = 4$;
- $a = 18$, $b = 4$. $a \,\%\, b = 2$, noile valori fiind $a = 4$, $b = 2$;

- $a = 4, b = 2$. $a \% b = 0$, noile valori fiind $a = 2, b = 0$;
- $a = 2, b = 0$. Deoarece $b = 0$, continuarea algoritmului ne-ar duce la împărțiri la 0, operație ce nu este validă.

Mai jos puteți găsi implementarea în C++ a CMMDC-ului și a CMMMC-ului, program ce află CMMDC și CMMMC pentru t perechi de numere. Complexitatea algoritmului este $O(\log n)$ pentru fiecare test.

```
#include <iostream>
using namespace std;

int gcd(int a, int b)
{
    while(b > 0)
    {
        int c = a % b;
        a = b;
        b = c;
    }
    return a;
}
int main()
{
    int t;
    cin >> t;
    for(int i = 1; i <= t; i++)
    {
        int a, b;
        cin >> a >> b;

        int cmmdc = gcd(a, b);
        long long cmmmc = 1LL * a / cmmdc * b;
```

```
        cout << cmmdc << " " << cmmmc << '\n';
    }
    return 0;
}
```

2.2. Lucrul cu divizorii unui număr

Definiția 2.2.1. *Un număr $n \geq 2$ este **număr prim** dacă și numai dacă are doar 2 divizori: 1 și n.*

Definiția 2.2.2. *Un număr $n \geq 2$ este **număr compus** dacă și numai dacă nu este prim.*

Observație. 0 și 1 nu sunt nici numere prime, nici numere compuse.

Observație. 2 este singurul număr prim par, celelalte numere prime fiind impare.

Definiția 2.2.3. *Descompunerea în factori primi se bazează pe Teorema fundamentală a aritmeticii, dată mai jos:*

Teorema 2.2.1 (Teorema fundamentală a aritmeticii)**.** *Orice număr natural $n > 1$ se poate scrie în mod unic sub forma $n = p_1^{e_1} \cdot p_2^{e_2} \cdot \cdots \cdot p_k^{e_k}$, unde $p_1 < p_2 < \cdots < p_k$ sunt numere prime, iar $e_1, e_2, \ldots, e_k$ sunt numere naturale nenule.*

Observație. Se poate observa că numărul maxim de numere prime la care se împarte un număr n este foarte mic(de exemplu, pentru $n \leq 10^9$, sunt cel mult 9 numere prime în reprezentarea ca produs de factori primi.

Pentru a afla divizorii unui număr natural n, cel mai simplu (dar și ineficient) algoritm constă în a verifica pe rând fiecare număr 1 la n și să verificăm dacă n se împarte exact la acel număr. Pentru a optimiza acest algoritm, va trebui să folosim o altă observație importantă.

Observație. Dacă n se împarte exact la x, se va împărți exact și la $\frac{n}{x}$. Asta ne duce la ideea să verificăm doar divizorii până la $\sqrt{n}$, observație ce se va dovedi fundamentală în calculele și algoritmii pe care îi vom scrie pentru toate aceste probleme.

Astfel, vom putea afla orice informație legată de divizorii unui număr în $O(\sqrt{n})$, fie că e vorba de numărul de divizori, divizorii primi, descompunerea în factori primi și așa mai departe.

Adauga sectiune despre techinici de identificarea rapida daca este divizibil sau nu

2.3. Problema divizibilitate de pe Kilonova

Se dă un număr t și t numere naturale. Să se afle pentru fiecare dintre ele răspunsul la una din următoarele întrebări:

- `1 n`: Să se afle dacă n este prim sau nu. În caz afirmativ se va afișa 'YES', altfel se va afișa 'NO'.
- `2 n`: Să se afle câți divizori are n — de exemplu, dacă $n = 12$, se va afișa 6 (1, 2, 3, 4, 6, 12 sunt divizorii lui 12).
- `3 n`: Să se afle numărul divizorilor primi ai lui n — de exemplu, dacă $n = 21$, se va afișa 2.
- `4 n`: Să se afișeze descompunerea în factori primi pe care o are un număr, fiecare factor fiind scris pe o linie, în ordine **crescătoare** a numerelor prime — de exemplu, dacă $n = 60$, se vor afișa pe 3 linii separate:

```
2 2
3 1
5 1
```

2. Divizibilitate

Fiecare tip de întrebare a fost implementat folosind o funcție separată pentru a arăta diferențele ce pot apărea de la un tip de întrebare la alta.

```
#include <iostream>
using namespace std;

void tip1(int n)
{
    bool prim = 1;
    if(n == 1)
        prim = 0;
    for(int i = 2; i * i <= n; i++)
        if(n % i == 0)
            prim = 0;
    if(prim)
        cout << "YES" << '\n';
    else
        cout << "NO" << '\n';
}

void tip2(int n)
{
    int nrdiv = 0;
    for(int i = 1; i * i <= n; i++)
        if(n % i == 0)
        {
            nrdiv++;
            if(n / i != i)
                nrdiv++;
        }
    cout << nrdiv << '\n';
}

void tip3(int n)
```

```
{
    int nrdivprim = 0;
    for(int i = 2; i * i <= n; i++)
        if(n % i == 0)
        {
            nrdivprim++;
            while(n % i == 0)
                n = n / i;
        }
    if(n > 1)
        nrdivprim++;
    cout << nrdivprim << '\n';
}

void tip4(int n)
{
    for(int i = 2; i * i <= n; i++)
        if(n % i == 0)
        {
            cout << i << " ";
            int cnt = 0;
            while(n % i == 0)
            {
                cnt++;
                n = n / i;
            }
            cout << cnt << '\n';
        }
    if(n > 1)
        cout << n << " " << 1 << '\n';
}

int main()
{
```

```
    int t;
    cin >> t;

    for(int i = 1; i <= t; i++)
    {
        int tip, n;
        cin >> tip >> n;

        // vom apela o alta functie pentru fiecare tip
        if(tip == 1)
            tip1(n);
        if(tip == 2)
            tip2(n);
        if(tip == 3)
            tip3(n);
        if(tip == 4)
            tip4(n);
    }

    return 0;
}
```

2.4. Probleme și lectură suplimentară

- Probleme cu divizibilitate de pe kilonova
- Number theory — Storing information about multiples/divisors
- Articol de pe USACO Guide

3. Stivă

3.1. Noțiuni introductive

Stiva este ca un teanc de obiecte. Ea are 4 operații principale:

1. *push(value)*: Adaugă *value* pe vârful stivei.
2. *top()*: Spune care este valoarea de pe vârful stivei.
3. *pop()*: Scoate elementul de pe vârful stivei.
4. *empty()*: Spune dacă stiva este goală.

Observație. Valorile vor fi returnate după regula *LIFO*, adică *last in, first out*.

3.2. Problema stiva

Această problemă ne cere să implementăm exact operațiile descrise mai sus. Aceasta este soluția:

```
#include <iostream>

using namespace std;

const int nmax = 1e4;
int stiva[nmax], lungime_stiva;
```

```
void push(int val)
{
    stiva[lungime_stiva] = val;
    lungime_stiva++;
}

int top()
{
    return stiva[lungime_stiva - 1];
}

void pop()
{
    lungime_stiva--;
}

bool empty()
{
    if (lungime_stiva == 0)
    {
        return 1;
    }
    return 0;
}

int main()
{
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++)
    {
        int cer;
        cin >> cer;
```

```
        if (cer == 0)
        {
            int val;
            cin >> val;
            push(val);
        }
        else if (cer == 1)
        {
            cout << top() << '\n';
        }
        else if (cer == 2)
        {
            pop();
        }
        else
        {
            cout << empty() << '\n';
        }
    }
    return 0;
}
```

Aceasta este soluția cu stiva din STL:

```
#include <iostream>
#include <stack>

using namespace std;

int main()
{
    stack<int> stiva;
    int n;
    cin >> n;
```

```
    for (int i = 1; i <= n; i++)
    {
        int cer;
        cin >> cer;
        if (cer == 0)
        {
            int val;
            cin >> val;
            stiva.push(val);
        }
        else if (cer == 1)
        {
            cout << stiva.top() << '\n';
        }
        else if (cer == 2)
        {
            stiva.pop();
        }
        else
        {
            cout << stiva.empty() << '\n';
        }
    }
    return 0;
}
```

3.3. Problema stiva_max_min

Da, aprob, faina problemă Traiane

4. Trie

4.1. Ce este un Trie

Un trie (sau arbore de prefixe) este un *arbore de căutare k-ar* (un arbore cu rădăcină unde fiecare nod are maxim k fii), reprezentând un mod unic de a memora informațiile, numite și *chei*.

Numărul de fii al unui nod este în mare parte influențat de tipul informațiilor memorate, dar de cele mai multe ori, un Trie este folosit pentru reținerea șirurilor de caractere, astfel fiecare nod având maxim 26 fii.

Inițial arborele conține doar un singur nod, rădăcina, urmând ca apoi cuvintele să fie introduse în ordinea citirii lor, de la stânga la dreapta. Observăm că înălțimea arborelui este lungimea maximă a unui cuvânt. Complexitatea de timp este O(lungime maximă), iar memoria consumată, în cel mai rău caz, este O(număr cuvinte $\cdot$ k).

4.2. Moduri de implementare

Există două modalități standard prin care putem implementa un Trie, folosind pointeri sau vectori. Ambele funcționează la fel de bine, însă operația de *delete* este mai greu de implementat cu vectori.

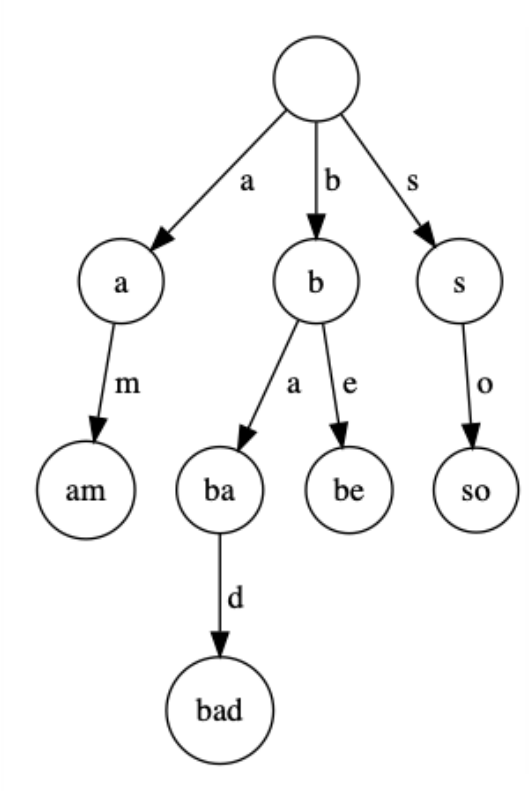


Figura 4.1.: Un exemplu de trie pentru cuvintele „am”, „bad”,„be” și „so”.

4.2.1. Prin pointeri

Ne vom folosi de o structură unde vom reține un contor reprezentând de câte ori am trecut prin nodul curent, cât și un vector de pointeri, reprezentând fiii nodului curent.

```
struct Trie {
    int cnt;
    Trie *fii[26];

    Trie() : cnt{0}, fii{nullptr} {}

    ~Trie() {
        for (auto &fiu: fii) delete fiu;
```

```
    }
};

Trie *root = new Trie;
```

Operația de *insert* poate fi foarte ușor scrisă recursiv.

```
void inserts(Trie *node, string a, int poz) {
    if (poz == a.size()) {
        node->cnt++;
        return;
    }

    if (node->fii[a[poz] - 'a'] == 0) {
        node->fii[a[poz] - 'a'] = new Trie;
    }

    inserts(node->fii[a[poz] - 'a'], a, poz + 1);
}
```

În momentul în care am ajuns la un nod *node* în arbore, verificăm dacă există fiul pentru caracterul următor și dacă nu există, îl adăugăm în arbore, apoi apelăm recursiv până ajungem la finalul stringului.

Pentru a elimina un string din trie ne mai trebuie o informație suplimentară, și anume să știm câți fii are un nod. Așadar, dacă am eliminat un sufix al șirului și nodul curent nu mai are fii nici nu mai este vizitat prin alt șir inserat, putem da erase complet la pointerul respectiv.

```
bool del(Trie *node, string a, int pos) {
    if (pos == a.size()) {
        node->cnt--;
    } else if (del(node->fii[a[pos] - 'a'], a, pos + 1)) {
        node->nrf--;
        node->fii[a[pos] - 'a'] = 0;
```

```
    }

    if (node->cnt == 0 && node->nrf == 0 && node != t) {
        delete node;
        return 1;
    }

    return 0;
}
```

Restul operațiilor se implementează similar, practic baza tuturor operațiilor stă în modul de a parcurge trie-ul.

4.2.2. Prin vectori

În loc de o structură vom folosi un vector cu *k* coloane. În fiecare element din vector vom reține poziția fiului respectiv.

```
vector<vector<int>> trie(1, vector<int>(26, -1));
```

Astfel *trie[node][5]* va fi egal cu poziția în vectorul `trie` pentru al cincilea fiu a lui *node*.

Operația de *insert* este foarte similară față de cea precedentă, singurul lucru care diferă este modul de implementare. În acest caz ne este mult mai ușor să folosim o funcție care să itereze propriu-zis prin șirul de caractere.

```
vector<vector<int>> trie(1, vector<int>(26, -1));
vector<int> cnt(1);

void insert(string a) {
    int root = 0;
    for (char i : a) {
        if (trie[root][i - 'a'] == -1) {
```

```
            trie[root][i - 'a'] = trie.size();
            trie.emplace_back(26, -1);
            cnt.push_back(0);
        }
        cnt[root]++;
        root = trie[root][i - 'a'];
    }
    cnt[root]++;
}
```

Observăm faptul că incrementăm și la final contorul.

4.3. Trie pe biți

Unele probleme necesită reținerea numerelor într-o structură de date, cum ar fi un trie, însa vom înlocui șirurile de caractere cu reprezentarea binară a numerelor.

4.3.1. Problema xormax de pe Kilonova (ușoară)

Un exemplu bun este chiar problema xormax, unde ni se dă un vector cu N elemente și trebuie să aflăm care este suma xor maximă a unui interval. Suma XOR a unui interval cu capetele $[L, R]$ este valoarea $v_L \oplus v_{L+1} \oplus \cdots \oplus v_R$, unde $\oplus$ este operatorul XOR pe biți.

Pentru a rezolva problema putem parcurge vectorul de la stânga la dreapta și să aflam pentru fiecare $1 \leq i \leq N$ care este suma XOR maximă a unui interval care se termină în i. Dacă construim vectorul xp, unde $xp[i] = v_1 \oplus v_2 \oplus \cdots \oplus v_{i-1} \oplus v_i$, atunci suma XOR pe intervalul $[L, R]$ este egală cu $xp[R] \oplus xp[L-1]$. Observăm că pentru un R fixat trebuie să găsim care este L-ul care maximizează relația de mai sus. Pentru a face asta putem să introducem pri-

mii $R - 1$ *xp*-uri într-un trie pe biți și să căutăm bit cu bit, începând cu bitul semnificativ, *xp*-ul care va maximiza rezultatul.

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

using namespace std;
const int N = 2e5 + 1;

vector<vector<int>> trie(1, vector<int>(2, -1));

int n, v[N], xp[N];

int find(int nr) {
    int root = 0;
    int ans = 0;
    for (int bit = 31; bit >= 0; bit--) {
        bool b = (nr & (1 << bit));
        if (trie[root][!b] == -1) {
            if (trie[root][b] == -1) {
                return ans;
            } else {
                root = trie[root][b];
            }
        } else {
            ans += (1 << bit);
            root = trie[root][!b];
        }
    }
    return ans;
}

void insert(int nr) {
    int root = 0;
```

```
    for (int bit = 31; bit >= 0; bit--) {
        bool b = (nr & (1 << bit));
        if (trie[root][b] == -1) {
            trie[root][b] = trie.size();
            trie.emplace_back(2, -1);
        }
        root = trie[root][b];
    }
}

int main() {
    cin.tie(0)->sync_with_stdio(0);
    cin >> n;

    for (int i = 1; i <= n; i++) {
        cin >> v[i];
        xp[i] = xp[i - 1] ^ v[i];
    }

    int ans = 0;
    insert(0);

    for (int i = 1; i <= n; i++) {
        int res = find(xp[i]);
        ans = max(ans, res);
        insert(xp[i]);
    }

    cout << ans;
}
```

O variantă care se folosește de implementarea cu pointeri este următoarea:

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

using namespace std;

struct Trie {
    Trie *_next[2];
    int _pos;

    explicit Trie(const int value)
            : _pos{value}, _next{nullptr, nullptr} {}

    Trie() : Trie{-1} {}

    ~Trie() {
        delete _next[0];
        delete _next[1];
    }
} *root;


void add(const int val, const int idx) {
    Trie *node = root;

    for (int i = 29; i >= 0; i--) {
        bool has = (val >> i) & 1;
        if (node->_next[has] == nullptr)
            node->_next[has] = new Trie(idx);
        node = node->_next[has];
    }
}

int query(const int val) {
    Trie *node = root;
```

```
    for (int i = 29; i >= 0; i--) {
        bool has = (val >> i) & 1;
        if (node->_next[!has])
            node = node->_next[!has];
        else if (node->_next[has])
            node = node->_next[has];
        else
            break;
    }
    return node->_pos;
}

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

    root = new Trie(0);

    int n, x, sum = 0, value = 0;
    cin >> n;
    vector<int> sums(n + 1);

    add(sum, 0);

    for (int i = 1; i <= n; i++) {
        cin >> x;
        sum ^= x;
        sums[i] = sum;

        value = max(value, x);

        if (i > 1) {
```

```
            int qry = query(sum);
            value = max(value, sum ^ sums[qry]);
        }

        add(sum, i);
    }

    cout << value;

    delete root;

    return 0;
}
```

4.4. Problema XOR Construction (medie)

În această problemă ni se dau $n-1$ numere, unde al i-lea are valoarea a_i, iar noi trebuie să construim alt vector b, cu n elemente, astfel încât să existe toate numerele de la 0 la $n-1$ în b, iar $b_i \oplus b_{i+1} = a_i$.

În primul rând, dacă $b_i = 0$ atunci $b_{i+1} = a_i$, $b_{i+1} \oplus b_{i+2} = a_{i+1}$, deci $b_{i+2} = a_i \oplus a_{i+1}$ și $b_{i+3} = a_i \oplus a_{i+1} \oplus a_{i+2}$. Prin urmare deducem o formă generală pentru b_j, unde $i < j$, și anume $b_j = a_i \oplus a_{i+1} \oplus a_{i+2} \oplus \cdots \oplus a_{j-1}$. Proprietatea se respectă și pentru oricare $j < i$, avem $b_j = a_j \oplus a_{j+1} \oplus \cdots \oplus a_{i-1}$.

În al doilea rând, enunțul problemei asigură faptul că mereu va exista soluție. Dar când nu avem soluție? Păi în momentul în care se repetă două elemente în vectorul b, ceea ce înseamnă faptul că trebuie să existe o secvență cu suma XOR egală cu 0. Pentru simplitate vom spune că pe poziția k va fi $b_k = 0$. Dacă $i < j$ și $b_i = b_j$ și $j < k$, atunci $a_i \oplus a_{i+1} \oplus a_{i+2} \oplus \cdots \oplus a_{j-1} = 0$, analog pentru $i > j > k$. Dacă $i < k < j$ și $b_i = b_j$ atunci $b_i = a_i \oplus a_{i+1} \oplus \cdots \oplus a_{k-1}$, $b_j = a_k \oplus a_{k+1} \cdots \oplus a_{j-1}$. Prin urmare $a_i \oplus a_{i+1} \oplus \cdots \oplus a_{j-1} = 0$. Așadar, știm

ca mereu în vectorul b elementele vor fi distincte.

În al treilea rând, observăm că vectorul b este generat în funcție de ce valoare are k. Deci o primă idee ar fi să fixăm mai întâi unde vom pune 0-ul în vectorul b și să-l construim în $O(n)$, complexitatea temporală fiind $O(n^2)$. Dar putem să ne folosim de a doua observație, și anume că mereu vectorul b va avea elementele distincte. Deci ne este suficient să știm care va fi valoarea maximă din b dacă 0-ul se află pe poziția k. Pentru a face asta putem să folosim 2 trie-uri, unul pentru sufix, altul pentru prefix, complexitatea finală devenind $O(n \log n)$.

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

using namespace std;
const int N = 2e5 + 1;

int n;
vector<int> v(N), ans(N);
vector<int> xr1(N), xr2(N);

vector<vector<int>> trie1(1, vector<int>(2, -1)),
                    trie2(1, vector<int>(2, -1));

vector<int> maxim1(N), maxim2(N);

void insert(vector<vector<int>> &trie, int nr) {
    int root = 0;
    for (int i = 30; i >= 0; i--) {
        bool bit = (nr & (1 << i));
        if (trie[root][bit] == -1) {
            trie[root][bit] = trie.size();
            trie.push_back(vector<int>(2, -1));
```

```
        }
        root = trie[root][bit];
    }

}

int get_max(vector<vector<int>> &trie, int nr) {
    int ans = 0;
    int root = 0;
    for (int i = 30; i >= 0; i--) {
        bool bit = (nr & (1 << i));
        if (trie[root][!bit] != -1) {
            ans += (1 << i);
            root = trie[root][!bit];
        } else if (trie[root][bit] != -1) {
            root = trie[root][bit];
        }
    }
    return ans;
}

int main() {
    cin >> n;
    int xr = 0;
    for (int i = 1; i < n; i++) {
        cin >> v[i];
        xr1[i] = xr1[i - 1] ^ v[i];
    }
    for (int i = n - 1; i >= 1; i--) {
        xr2[i] = xr2[i + 1] ^ v[i];
    }

    maxim1[1] = 0;
    maxim2[n] = 0;
```

```
insert(trie1, xr2[1]);
for (int i = 2; i <= n; i++) {
    maxim1[i] = get_max(trie1, xr2[i]);
    insert(trie1, xr2[i]);
}
insert(trie2, xr1[n - 1]);
for (int i = n - 2; i >= 0; i--) {
    maxim2[i] = get_max(trie2, xr1[i]);
    insert(trie2, xr1[i]);
}


for (int i = 1; i <= n; i++) {
    if (max(maxim1[i], maxim2[i - 1]) == n - 1) {
        int xr1 = 0, xr2 = 0;
        vector<int> fr(2 * n + 1);
        fr[0] = 1;
        ans[i] = 0;
        for (int j = i - 1; j >= 1; j--) {
            ans[j] = v[j] ^ ans[j + 1];
            xr1 ^= v[j];
            fr[ans[j]]++;
            if (fr[ans[j]] >= 2) {
                break;
            }
        }
        for (int j = i; j < n; j++) {
            ans[j + 1] = v[j] ^ xr2;
            xr2 ^= v[j];
            fr[ans[j + 1]]++;
            if (fr[ans[j + 1]] >= 2) {
                break;
            }
        }
```

```
            int ok = 1;
            for (int j = 0; j < n; j++) {
                if (fr[j] != 1) {
                    ok = 0;
                    break;
                }
            }
            if (1) {
                for (int j = 1; j <= n; j++) {
                    cout << ans[j] << " ";
                }
                return 0;
            }
        }
    }
}
```

4.5. Problema cuvinte (medie-grea)

Se dau N cuvinte formate doar din primele K litere mici ale alfabetului englez și un șir x_i, de M numere naturale. Trebuie să se formeze M cuvinte astfel încât oricare cuvânt ($1 \leq i \leq M$) să respecte următoarele proprietăți:

- Să aibă lungimea x_i.
- Să fie format doar din primele K litere mici ale alfabetului englez.
- Să nu existe $j \leq M, j \neq i$, sau un cuvânt *cuv* din cele N, astfel încât cuvântul j să fie prefix pentru cuvântul i, sau *cuv* să fie prefix pentru i.
- Să nu existe $j \leq M, j \neq i$, sau un cuvânt *cuv* din cele N, astfel încât cuvântul i să fie prefix pentru cuvântul j, sau i să fie prefix pentru *cuv*.

Prima idee ar fi să sortam vectorul x. Fie dp_i = în câte moduri putem alege primele i cuvinte. Putem considera toate posibilitățile de a forma șirurile , iar abia apoi să vedem cum eliminăm pe cele care nu sunt bune. Cu alte cuvinte, fie $(s_1, s_2, .., s_{i-1})$ primele $i-1$ cuvinte alese astfel încât să respecte condițiile impuse de problemă. Sunt în total $dp_{i-1} \cdot K^{x_i}$ moduri de a forma un set de șiruri cu primele i cuvinte.

Observație. Nu există două cuvinte, s_x și s_y, astfel încât ambele să fie prefixe pentru s_i.

Demonstrație. Dacă ambele ar fi prefixe pentru s_i, atunci fie s_x este prefix pentru s_y, fie invers, ceea ce este fals, pentru că noi am generat primele $i-1$ cuvinte optim. □

Astfel dacă pentru fiecare cuvânt k, $k < i$, putem să scădem din numărul total de posibilități șirurile unde s_k este prefix pentru s_i, nu vom elimina două configurații la fel. $dp_i = dp_{i-1} \cdot K^{x_i} - dp_{i-1} \cdot \sum_{j=1}^{i-1} K^{x_i - x_j}$.

Observație. Nu există două cuvinte, unul provenit din cele N date și celălalt (s_k) din primele $i-1$ astfel încât ambele să fie prefixe pentru s_i.

Demonstrație. Dacă ambele sunt prefixe pentru s_i, atunci fie s_k este prefix pentru un cuvânt din cele N, fie invers. □

Deci, putem să fixam un cuvânt din cele N date inițial și să eliminăm numărul de posibilități ca el să fie prefix pentru s_i. Datorită observației, nu vom elimina o posibilitate dacă a fost eliminată deja în prima etapă. În mod natural vom zice că din dp-ul nostru vom scădea în mod similar $dp_{i-1} \cdot \sum_{j=1}^{N} K^{x_i - len(j)}$, unde $len(j)$ = lungimea cuvântului j, cu $x_i \geq len(j)$. Însă nu este adevărat, pentru că dacă avem două cuvinte x și y , unde x este prefix pentru y, atunci suma de mai sus va număra 2 configurații de două ori. Observăm că nouă

ne trebuie practic doar acele cuvinte x, pentru care nu există alt cuvânt y, cu y prefix pentru x, iar $len(x) \leq x_i$. Astfel putem parcurge direct pe Trie-ul cuvintelor. Dacă suntem la un nod *node*, acesta este capătul unui cuvânt, iar $len(cuv) \leq x_i$, atunci putem scădea din dp-ul nostru $dp_{i-1} \cdot K^{x_i - len(cuv)}$ și să oprim parcurgerea. Dacă suntem la un nod *node*, acesta are lungimea egală cu x_i, atunci scădem din dp dp_{i-1} și oprim parcurgerea.

Cu alte cuvinte, o soluție în $O(M^2 + M \cdot S)$ este posibilă, unde $S = \sum_{i=1}^{N} len(i)$. Putem optimiza soluția, observând că de fiecare dată putem face tranzițiile în $O(1)$. Soluția finală devine $O(M + S)$ sau $O(M \cdot \log + S)$.

```
#include <bits/stdc++.h>
using namespace std;
const int mod = 1e9 + 7, N = 3e5 + 1;
struct Mint
{
    int val;
    Mint(int x = 0)
    {
        val = x % mod;
    }
    Mint(long long x)
    {
        val = x % mod;
    }
    Mint operator+(Mint oth)
    {
        return val + oth.val;
    }
    Mint operator*(Mint oth)
    {
        return 1LL * val * oth.val;
    }
    Mint operator-(Mint oth)
```

```
    {
        return val - oth.val + mod;
    }
    Mint fp(Mint a, long long n){
        Mint p = 1;
        while(n){
            if(n & 1){
                p = p * a;
            }
            a = a * a;
            n /= 2;
        }
        return p;
    }
    Mint operator/(Mint oth){
        Mint invers = fp(oth, mod - 2);
        return 1LL * val * invers.val;
    }
    friend ostream& operator << (ostream& os, const Mint& lol){
        os << lol.val;
        return os;
    }
};

int n, m, k;
vector<Mint> dp(N);
vector<int> x(N), depth(N), cnt1(N);
vector<vector<int>> trie(1, vector<int>(26,-1));
vector<bool> cnt(1);
Mint spm = 0;
Mint fp(Mint a, int n){
    Mint p = 1;
    while(n){
        if(n & 1) p = a * p;
```

```
        a = a * a;
        n /= 2;
    }
    return p;
}

void insert(string a){
    int root = 0;
    for(int i = 0; i < a.size(); i++){
        if(trie[root][a[i]-'a'] == -1){
            trie[root][a[i]-'a'] = trie.size();
            trie.push_back(vector<int>(26, -1));
            cnt.push_back(0);
        }
        root = trie[root][a[i]-'a'];
    }
    cnt[root]=1;
}
void dfs(int node, int lenx, int len){
    if(lenx == len){
        return;
    }
    if(cnt[node]){
        spm = spm + fp(k, lenx - len);
        return;
    }
    for(int i = 0; i < 26; i++){
        if(trie[node][i] != -1){
            dfs(trie[node][i], lenx, len + 1);
        }
    }
}
void dfs1(int node, int len){
    depth[len]++;
```

```
    if(cnt[node]){
        cnt1[len]++;
        return;
    }
    for(int i = 0; i < 26; i++){
        if(trie[node][i] != -1){
            dfs1(trie[node][i], len+1);
        }
    }
}
int main(){
    cin.tie(0)->sync_with_stdio(0);
    cin >> n >> m >> k;
    for(int i = 1; i <= n; i++){
        string a;
        cin >> a;
        insert(a);
    }
    for(int i = 1; i <= m; i++){
        cin >> x[i];
    }
    sort(x.begin() + 1, x.begin() + 1 + m);
    dp[1] = fp(k, x[1]);
    Mint sm = 0;
    dfs(0, x[1], 0);
    dfs1(0, 0);
    dp[1] = dp[1] - depth[x[1]];
    dp[1] = dp[1] - spm;
    for(int i = 2; i <= m; i++){
        dp[i] = dp[i - 1] * fp(k, x[i]);
        sm = sm * fp(k, x[i]-x[i-1]);
        sm = sm + fp(k, x[i]-x[i-1]);
        // for(int j = i - 1; j >= 1; j--){
        //     dp[i] = dp[i] - dp[i-1]*fp(k, x[i]-x[j]);
```

```
        // }
        dp[i] = dp[i] - dp[i-1]*sm;
        spm = spm * fp(k, x[i]-x[i-1]);
        for(int j = x[i-1]; j < x[i]; j++){
            spm = spm + fp(k, x[i] - j) * cnt1[j];
        }
        dp[i] = dp[i] - dp[i-1]*depth[x[i]];
        dp[i] = dp[i] - dp[i-1]*spm;
        //dfs(0, x[i], 0, dp[i], dp[i - 1]);
    }
    cout << dp[m];
}
```

4.6. Problema cli (medie-grea)

Se dau N cuvinte care trebuie tastate într-un terminal. Un cuvânt este considerat tastat dacă el va apărea în terminal cel puțin odată pe parcursul tastării. Avem două tipuri de operații la dispoziție: adăugăm un caracter la finalul șirul tastat deja, eliminăm un caracter de la finalul șirului (dacă nu este vid). Pentru fiecare $i = \overline{1, K}$, noi trebuie să aflam care este numărul minim de operații pentru a tasta exact i cuvinte distincte dintre cele date. În momentul în care începem să tastăm un cuvânt, trebuie mereu să începem de la un șir vid[1], și să terminăm tastarea tot la un șir vid. Un exemplu de tastare corectă este: $\emptyset \longrightarrow a \longrightarrow ab \longrightarrow abc \longrightarrow ab \longrightarrow a \longrightarrow \emptyset$.

Ne vom folosi din nou de metoda programării dinamice, dar de data asta vom face dp direct pe trie. Astfel, fie $dp[nod][i]$ = numărul minim de operații pentru a tasta i cuvinte cu prefixul format din lanțul de la rădăcină la nod. Acum, pentru un nod fixat din trie-ul nostru, putem presupune că în momentul tastării vom începe mereu cu șirul format de la rădăcină la nod, în loc de

[1]Voi nota șirul vid cu $\emptyset$. *n.red.*

$\emptyset$. De exemplu, dacă cuvintele au prefixul *abab*, atunci noi vom presupune o succesiune validă de operații: $abab \rightarrow abab\mathbf{c} \rightarrow \cdots \rightarrow abab\mathbf{c} \rightarrow abab$. Putem deci face un rucsac pentru fiii nodului, $dp1[i][j]$ = care e numărul minim de operații pentru a tasta j cuvinte din primii i fii. Pentru că prefixul necesită $\text{len}(prefix)$ operații de adăugare și ștergere, vom începe dp-ul nostru cu $2 \cdot \text{len}(prefix)$ operații deja făcute. Cu alte cuvinte, pentru a tasta 0 cuvinte vom face $dp1[0][0] = 2 \cdot \text{len}(prefix)$. În momentul în care trecem de la i la $i+1$ avem 2 cazuri: fie nu luăm fiul respectiv în considerare, fie alegem p șiruri pe care le vom tasta în $dp[fiu(i)][p] - 2 \cdot \text{len}(prefix)$ operații.

```
for (int i = 1; i <= 26; i++){
    for (int k1 = 0; k1 <= min(sz[nod], k); k1++){
        dp1[i][k1] = min(dp1[i][k1], dp1[i-1][k1]);
        for (int k2 = 1; k2 <= k1 && trie[nod][i-1] != -1
                                  && k2 < dp[trie[nod][i-1]].size(); k2++){
            dp1[i][k1] = min(dp1[i][k1],
                             dp1[i-1][k1-k2] + dp[trie[nod][i-1]][k2] - 2*len);
        }
    }
}
```

Problema constă în faptul că secvența de cod de mai sus rulează pentru fiecare nod din trie, ceea ce ar rezulta într-o complexitate de $O(N \cdot K^2)$. Doar că, în practică soluția are complexitatea de $O(N \cdot K)$. În momentul în care facem rucsac pe un arbore, este foarte important să fim atenți la memoria și la timpul consumate. Observăm faptul că cele două bucle merg până la $\min(sz[nod], k)$, lucru ce îmbunătățește timpul de execuție considerabil. Puteți citi mai multe din soluția problemei *Barricades*, iar sursa completă o puteți vizualiza aici.

4.7. Probleme

1. intervalxor2 (*Trie pe biți persistent. Puteți face queriurile și offline*)
2. xortree2 (*Problemă **ok** cu trie pe biți*)
3. Rps (*Alt exemplu de dp pe trie*)
4. ratina (*LCA pe trie*)
5. aiacupalindroame (*Nuj ce-i pbma asta, am pus-o că are trie la taguri*)
6. Facebook Search (*Sugestiv numele*)
7. Perfect Security
8. Collapsing Strings (*Dc e asta E?*)

5. Introducere în grafuri

În cele ce urmează voi prezenta o structură de date cu foarte multe aplicații atât în algoritmică, cât și în viața de zi cu zi, acestea fiind grafurile. Problema aflării existenței unor conexiuni sau aflării distanței minime între două noduri reprezintă un punct de plecare pentru majoritatea algoritmilor pe grafuri, teoria folosită în algoritmică fiind una vastă și plină de abordări ce se dovedesc a fi esențiale în foarte multe situații, atât competiționale, cât și în aplicații practice.

5.1. Noțiuni introductive

Definiția 5.1.1. *Un graf este o structură care corespunde unui grup de obiecte, în care unele perechi de obiecte sunt într-un anumit sens „legate" reciproc. Obiectele corespund unor abstracții matematice numite într-un graf noduri/vârfuri (numite și puncte) și fiecare legătură dintre perechile de obiecte asociate se numește muchie (numită și arc sau linie, prin care este și reprezentată). De obicei, un graf este reprezentat în formă schematică ca un set/grup de puncte pentru noduri, iar acestea sunt unite două câte două de drepte sau curbe pentru muchii.*

5.1.1. Terminologie

Voi continua prin a defini termeni ce se dovedesc a fi esențiali pentru înțelegerea grafurilor.

Definiția 5.1.2. *Definim un **graf neorientat** ca fiind un graf pentru care dacă avem o muchie (A, B), o putem folosi pentru a ajunge atât de la A la B, cât și de la B la A.*

Definiția 5.1.3. *Definim un **graf orientat** ca fiind un graf pentru care dacă avem o muchie (A, B), o putem folosi **doar** pentru a ajunge atât de la A la B, nu și de la B la A.*

Definiția 5.1.4. *Două noduri sunt **adiacente** atunci când există cel puțin o muchie care să le lege direct.*

Definiția 5.1.5. *Folosim noțiunea de **incidență** când spunem că un nod este extremitate a unei muchii.*

Definiția 5.1.6. *Definim **gradul** unui nod ca fiind numărul de muchii incidente cu acel nod.*

Observație. Suma gradelor nodurilor într-un graf neorientat este mereu un număr par. Explicația este dată de faptul că pentru fiecare muchie adăugată, gradul a două noduri crește cu 1.

Definiția 5.1.7. *Numim **lanț** o secvență de noduri ce au proprietatea că oricare două vecine reprezintă capetele unei muchii a grafului. Se disting noțiunile de **lanț elementar** (lanț cu nodurile distincte) și **lanț simplu** (lanț cu muchiile distincte).*

Definiția 5.1.8. *Un **ciclu** reprezintă o secvență de muchii ce nu se repetă, pleacă de la un nod A și parcurgând în ordine acele muchii, se ajunge tot la nodul A. Din nou, se distinge noțiunea de **ciclu simplu** ca fiind un ciclu în care nu se repetă noduri.*

Definiția 5.1.9. *Definim **lungimea unui lanț** ca fiind numărul de muchii folosite pentru a ajunge de la un capăt al lanțului la celălalt. Uneori, lungimea se definește ca numărul de noduri prin care trecem, numărându-se un nod de câte ori s-a trecut prin el.*

Observație. În orice caz, lungimea exprimată prin numărul de noduri este cu 1 mai mare decât lungimea exprimată prin numărul de muchii.

Definiția 5.1.10. *Definim **graf parțial** al unui graf dat ca fiind ceea ce rămâne din graful dat păstrând toate nodurile și eliminând eventual unele muchii, fără a adăuga muchii noi.*

Definiția 5.1.11. *Definim **subgraf** al unui graf dat ca fiind ceea ce rămâne din graful dat eliminând unele noduri și doar muchiile incidente lor, deci nu și alte muchii și fără să adăugăm alte muchii.*

Observație. Numărul de subgrafuri ale unui graf este 2^n, iar numărul de grafuri parțiale este 2^m, unde n este numărul de noduri, iar m este numărul de muchii al grafului.

5.1.2. Câteva tipuri speciale de grafuri

Se remarcă faptul că în funcție de tipul grafului, mai putem defini următoarele tipuri de grafuri, care se vor folosi în diferite aplicații. De notat ca pentru unele din aceste tipuri, vom avea probleme unde vom explica în detaliu noțiunile și aplicațiile unde folosim aceste concepte.

Definiția 5.1.12. *Definim un **graf complet** ca fiind un graf care are toate muchiile posibile, existând o legătură directă de la* (A, B) $\forall 1 \leq A < B \leq n$. *Numărul de muchii ale unui graf complet cu n noduri este* $\frac{n(n-1)}{2}$.

Definiția 5.1.13. *Definim un **graf bipartit** ca fiind un graf care poate fi împărțit în două submulțimi disjuncte, A și B, astfel încât nu există nicio muchie între două elemente din aceeași submulțime.*

Definiția 5.1.14. *Definim un **graf planar** ca fiind un graf care are proprietatea că poate fi reprezentat grafic fără ca două muchii să se intersecteze.*

Definiția 5.1.15. *Definim un **graf regulat** ca fiind un graf care are proprietatea că toate nodurile au același grad.*

5.2. Lucrul cu grafuri. Moduri de reprezentare în memorie

Un concept foarte important în teoria grafurilor reprezintă modul în care parcurgem aceste structuri de date și cum putem verifica proprietățile de care avem nevoie, de la o problemă la alta.

Să considerăm graful neorientat din figura 5.1. Acest graf are 13 noduri și 12 muchii, acestea fiind (1, 4), (1, 3), (4, 9), (9, 3), (4, 2), (4, 6), (2, 6), (2, 5), (8, 12), (8, 11), (8, 10), (8, 7).

Pentru a reprezenta un graf în memorie, există trei moduri principale de a o face, cu distincția că în practică se va folosi doar reprezentarea prin liste de vecini.

Definiția 5.2.1. *Definim **matricea de adiacență a unui graf** ca fiind o matrice binară pentru care $a_{ij} = 1$ dacă și numai dacă avem muchie de la nodul i la nodul j și $a_{ij} = 0$ în caz contrar.*

Observație. Pentru un graf neorientat, matricea este mereu simetrică, adică $a_{ij} = a_{ji}\ \forall i, j$.

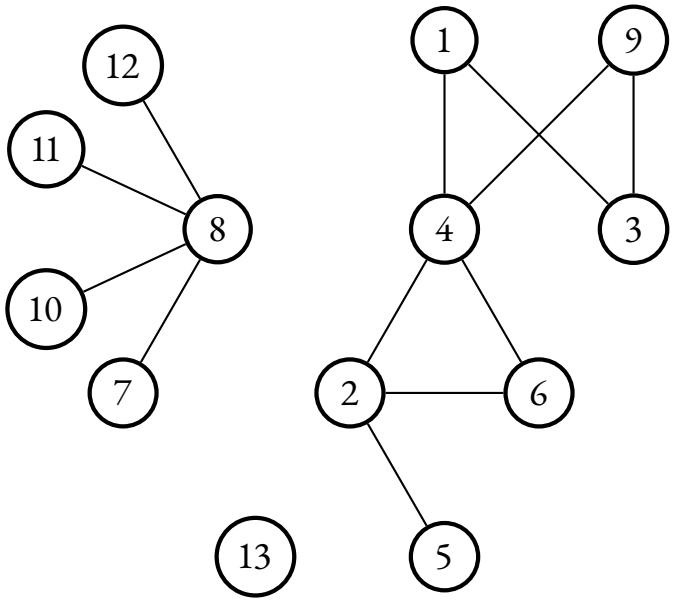


Figura 5.1.: Un graf neorientat cu 13 noduri și 12 muchii.

Pentru graful nostru de la fig. 5.1, aceasta este matricea de adiacență la care ajungem.

$$\begin{pmatrix}
0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\
\mathbf{1} & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\
0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \mathbf{1} & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 \\
0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

Definiția 5.2.2. *Definim o* ***lista de vecini*** *ca fiind o listă (de regulă, alocată*

dinamic) pe care o folosim pentru a ține în memorie pentru fiecare nod doar nodurile adiacente cu acesta, această metodă fiind cea mai eficientă din punct de vedere practic pentru a parcurge grafurile.

Noduri	**Vecini**
1	{3, 4}
2	{4, 5, 6}
3	{1, 9}
4	{1, 2, , 9}
5	{2}
6	{2, 4}
7	{8}
8	{7, 10, 11, 12}
9	{3, 4}
10	{8}
11	{8}
12	{8}
13	∅

Tabela 5.1.: Lista de vecini pentru graful neorientat din figura 5.1

Definiția 5.2.3. *Definim o **lista de muchii** ca fiind o listă pe care o folosim pentru a ține toate muchiile în memorie. Deși nu este o variantă prea practică de a efectua parcurgerile, această metodă poate fi utilă pentru anumiți algoritmi ce se bazează în principal pe prelucrarea muchiilor, un astfel de exemplu fiind arborele parțial de cost minim.*

Observație. În cazul nostru, lista de muchii este:

$$\{\{1, 4\}, \{1, 3\}, \{4, 9\}, \{9, 3\}, \{4, 2\}, \{4, 6\},$$
$$\{2, 6\}, \{2, 5\}, \{8, 12\}, \{8, 11\}, \{8, 10\}, \{8, 7\}\}$$

5.3. Conexitate. Parcurgerea DFS

Problema aflării conexității unui graf este una din problemele fundamentale ale teoriei grafurilor, fiind adesea folosită drept un exemplu esențial în explicarea și înțelegerea grafurilor.

Definiția 5.3.1. *Definim un **graf conex** ca fiind un graf neorientat care are proprietatea că pentru oricare două noduri A și B din graf, putem ajunge de la A la B folosind una sau mai multe muchii.*

Definiția 5.3.2. *Definim o **componentă conexă** ca fiind un subgraf **conex** al unui graf dat.*

Pentru a rezolva problema aflării conexității unui graf, va trebui să parcurgem graful folosind unul din algoritmii consacrați pentru această problemă. În cazul de față, vom continua prin a explica parcurgerea în adâncime a grafului (DFS sau depth-first search), una din parcurgerile optime pentru această problemă.

Definiția 5.3.3. *Definim **parcurgerea în adâncime** (DFS, engl. „depth-first search") a unui graf ca fiind o parcurgere recursivă ce pleacă de la un nod anume, iar la fiecare pas, dacă ne aflăm la un nod x, vom vizita vecinii nevizitați ai nodului x, apelând DFS pentru fiecare din ei.*

Observație. Complexitatea parcurgerii în adâncime (DFS) este $O(|V| + |E|)$, unde $|V|$ reprezintă numărul de noduri sau vârfuri și $|E|$ reprezintă numărul de muchii.[1].

Observație. Se remarcă faptul că un nod va fi vizitat la un moment dat doar o singură dată, deci dacă avem muchiile (1, 2), (1, 3) și (2, 3), iar DFS-ul pleacă din 1, 2 va fi accesat din 1, iar 3 va fi accesat din 2.

[1]În probleme se notează convențional $|V|$ cu N de la noduri, respectiv $|E|$ cu M de la muchii. *n.red.*

Observație. Se poate remarca faptul că ordinea în care vizităm nodurile în graf depinde de ordinea în care sunt adăugate muchiile în graf, acest lucru înseamnă că nu putem folosi DFS pentru anumite probleme, de exemplu cele la care trebuie aflată distanța minimă în graf.

5.4. Problema Connected components de pe kilonova

Se dă un graf neorientat G cu N noduri și M muchii. Să se afle câte componente conexe are graful dat.

Pentru a afla numărul de componente conexe ale unui graf, putem folosi parcurgerea DFS pentru a afla toate nodurile din care apelăm DFS din funcția *main*, acesta fiind și răspunsul la problema noastră.

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

using namespace std;

int n, m;

vector <vector<int> > v;
bool visited[100002];
int cc;
void dfs(int node)
{
    visited[node] = true;
    for(int i = 0; i < v[node].size(); ++i)
    {
        int nxt = v[node][i];
        if(visited[nxt] == false)
```

```
            dfs(nxt);
    }
}
int main()
{
    cin >> n >> m;
    v.resize(n+1);
    for(int i = 1; i <= m; ++i)
    {
        int a, b;
        cin >> a >> b;
        v[a].push_back(b);
        v[b].push_back(a);
    }
    for(int i = 1; i <= n; ++i)
        if(visited[i] == false)
        {
            ++cc;
            dfs(i);
        }

    cout << cc << '\n';
    return 0;
}
```

5.5. Drumuri minime. Parcurgerea BFS

Dacă în cazul parcurgerii DFS putem să o aplicăm fără mari probleme pentru o varietate destul de largă de probleme cu grafuri, totuși nu este suficientă pentru problemele ce țin de distanțe. Un exemplu fundamental este acela al aflării drumului minim între două sau mai multe noduri într-un graf dat.

Definiția 5.5.1. *Definim un **drum minim** ca fiind lungimea minimă a unui lanț care leagă două noduri din graf.*

Motivul pentru care nu putem afla drumul minim între două noduri folosind DFS este acela că ordinea în care nodurile sunt parcurse în DFS depinde de ordinea în care sunt date muchiile de la intrare, parcurgerea recursivă făcând aflarea distanțelor minime imposibilă. Astfel, vom introduce un alt mod de a parcurge graful nostru.

Definiția 5.5.2. *Definim **parcurgerea în lățime** (BFS, engl. „breadth-first search") a unui graf ca fiind o parcurgere iterativă ce pleacă de la unul sau mai multe noduri, iar la fiecare pas, dacă ne aflăm la un nod x, vom vizita vecinii nevizitați ai nodului x, adăugându-i într-o coadă, nodurile fiind parcurse în ordinea în care au fost adăugate în coadă.*

Observație. Complexitatea parcurgerii în lățime (DFS) este $O(|V| + |E|)$, unde $|V|$ reprezintă numărul de noduri sau vârfuri și $|E|$ reprezintă numărul de muchii.

Observație. Se poate remarca faptul că ordinea în care vizităm nodurile în graf va fi aceeași cu ordinea crescătoare a distanței minime față de nodul sau nodurile inițiale, datorită faptului că ele vor fi inserate în coadă în ordinea în care acestea au fost adăugate.

5.6. Problema Simple Shortest Path de pe kilonova

Se dă un graf neorientat G cu N noduri și M muchii, precum și un nod S. Să se afle lungimea drumului minim dintre S și fiecare nod din graf, inclusiv S.

Pentru a rezolva această problemă, vom pleca cu un BFS din nodul S și vom afla pe parcurs, distanțele minime față de toate celelalte noduri.

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

using namespace std;

int n, m, s;

vector<vector<int> > graf;
queue<int> q;
vector<int> ans;

int main()
{
    cin >> n >> m >> s;
    graf.resize(n+1);
    ans.resize(n+1);
    for(int i = 1; i <= m; i++)
    {
        int a, b;
        cin >> a >> b;
        graf[a].push_back(b);
        graf[b].push_back(a);
    }
    for(int i = 1; i <= n; i++)
        ans[i] = -1;
    ans[s] = 0;

    q.push(s);
    while(!q.empty())
    {
```

```
        int nod = q.front();
        q.pop();
        for(auto x : graf[nod])
        {
            if(ans[x] == -1)
            {
                ans[x] = ans[nod] + 1;
                q.push(x);
            }
        }
    }

    for(int i = 1; i <= n; i++)
        cout << ans[i] << " ";
    cout << '\n';

    return 0;
}
```

5.7. Problema grarb de pe infoarena

Se dă un graf G neorientat cu N noduri numerotate de la 1 la N și M muchii. Determinați numărul minim de muchii care trebuie eliminate și numărul minim de muchii care trebuie adăugate în graful G astfel încât acesta sa devina arbore.

Această problemă se împarte în două subprobleme relativ ușor de identificat - aflarea componentelor conexe ale grafului (dacă avem nr componente conexe, va fi nevoie de $nr-1$ muchii pentru a transforma graful într-unul conex), precum și aflarea numărului de muchii care trebuie scoase pentru a transforma graful în arbore (la final, trebuie să ne rămână $N-1$ muchii). Astfel, vom avea nevoie de $nr-1$ muchii noi și va trebui să scoatem $M+nr-1-(N-1)$

muchii pentru a avea un arbore.

```
#include <fstream>
#include <vector>

using namespace std;

ifstream cin("grarb.in");
ofstream cout("grarb.out");

int n, m, nr = 0;
vector<vector<int> > v;
vector<int> viz;

void dfs(int nod)
{
    viz[nod] = 1;
    for(int i = 0; i < (int) v[nod].size(); i++)
    {
        int nxt = v[nod][i];
        if(!viz[nxt])
            dfs(nxt);
    }
}
int main()
{
    cin >> n >> m;
    v.resize(n+1);
    viz.resize(n+1);

    for(int i = 1; i <= m; i++)
    {
        int a, b;
        cin >> a >> b;
```

```
        v[a].push_back(b);
        v[b].push_back(a);
    }

    for(int i = 1; i <= n; i++)
        if(!viz[i])
        {
            dfs(i);
            nr++;
        }

    cout << m + nr - 1 - (n - 1) << '\n' << nr - 1 << '\n';
    return 0;
}
```

5.8. Problema graf de pe kilonova

Se dă un graf neorientat conex cu N noduri și două noduri X și Y, să se afle nodurile ce aparțin tuturor lanțurilor optime între X și Y.

Pentru a rezolva această problemă, va trebui mai întâi să aflăm folosind o parcurgere de tip BFS distanțele minime de la X și Y spre toate celelalte noduri. Apoi, pentru fiecare distanță d de la 0 la dist(X, Y), vrem să aflăm câte noduri se află pe unul din drumurile optime de la X la Y la o distanță d față de X. În cele din urmă, vrem să afișăm nodurile situate la distanțele care apar o singură dată în mulțimea nodurilor ce fac parte din cel puțin un drum optim de la X la Y.

Codul sursă se poate viziona mai jos.

```
#include <fstream>
#include <vector>
#include <queue>
```

```
using namespace std;

const int MAXN = 7500;

int distX[MAXN + 1], distY[MAXN + 1], solFreq[MAXN + 1];

vector<vector<int>> goFromTo(MAXN + 5);
vector<int> ans;
queue<int> q;

void bfs(int nod, int dist[])
{
    int first;
    q.push(nod);
    dist[nod] = 1;

    while(!q.empty())
    {
        first = q.front();
        q.pop();

        for(auto travelNod : goFromTo[first])
        {
            if(dist[travelNod] == 0)
            {
                dist[travelNod] = dist[first] + 1;
                q.push(travelNod);
            }
        }
    }
}

int main()
{
```

```
ifstream cin("graf.in");
ofstream cout("graf.out");

int n, m, x, y, i, a, b, ansN;
cin >> n >> m >> x >> y;
for(i = 1; i <= m; i++)
{
    cin >> a >> b;
    goFromTo[a].push_back(b);
    goFromTo[b].push_back(a);
}

bfs(x, distX);
bfs(y, distY);

for(i = 1; i <= n; i++)
{
    // lungimea totala a drumului va fi egala cu distX[y];
    if(distX[i] + distY[i] == distX[y] + 1)
        solFreq[distX[i]]++;
}

ansN = 0;
for(i = 1; i <= n; i++)
    if(distX[i] + distY[i] == distX[y] + 1 && solFreq[distX[i]] == 1)
    {
        ansN++;
        ans.push_back(i);
    }

cout << ansN << '\n';
for(i = 0; i < ansN; i++)
    cout << ans[i] << ' ';
return 0;
```

```
}
```

5.9. Probleme și lectură suplimentară

- Probleme cu grafuri de pe kilonova
- Grafuri - noțiuni teoretice de bază
- Articol introductiv de pe USACO Guide
- Articol despre parcurgeri de pe USACO Guide
- Probleme cu grafuri de pe codeforces, ordonate după dificultate

Partea III.

Avansați