

Tutorial - NATPRO Teorija brojeva

1. GCD – najveći zajednički djelitelj

GCD je najveći broj s kojim su dva zadana broja djeljiva. Za njegovo nalaženje koristimo Euklidov algoritam.

Neke implementacije:

```
//straightfoward implementacija -> mana sporst
int gcd(int a, int b) {
    int sol = 1;
    for (int x = 2; x <= a && x <= b; ++x)
        if (a % x == 0 && b %x == 0)
            sol = x;
    return sol;
}

//rekurzivna implementacija s modanjem
int gcd(int a, int b) {
    int t;
    while (b) {
        if (a >= b)
            a = a % b;
        else
            t = b; b = a; a = t;
    }
    return a;
}

//najkraca implementacija – najcesce se koristi
int gcd(int a, int b) {
    return b ? gcd(b, a % b) : a;
}
```

2. Modularna aritmetika - predstavlja aritmetički sustav u kojem se brojevi „vrte u krug“ nakon što prijeđu određenu vrijednost (MOD).

U modularnoj aritmetici vrijede sljedeća pravila:

1. $(a \% \text{MOD} + b \% \text{MOD}) \% \text{MOD} == (a+b) \% \text{MOD}$
2. $(a \% \text{MOD} * b \% \text{MOD}) \% \text{MOD} == (a*b) \% \text{MOD}$
3. $(a \% \text{MOD} - b \% \text{MOD}) \% \text{MOD} == (a-b) \% \text{MOD}$
4. $(a * b^{(\text{MOD}-2)}) \% \text{MOD} == (a/b) \% \text{MOD}$, pri čemu MOD mora biti prost broj.
a, b, MOD $\in \mathbb{Z}$ (skup cijelih brojeva).

3. Linearne diofantske jednačbe – jednačbe oblika:

$$a*x + b*y = c, \quad a, b, c, x, y \in \mathbb{Z}.$$

One imaju rješenje ako $\text{gcd}(a, b) \mid c$. U slučaju da su a i b relativno prosti, jednačbu je moguće riješiti na način da se proširenim Euklidovim algoritmom izračuna rješenje temeljne jednačbe

$$a*x' + b*y' = 1,$$

te je onda :

$$x = c*x'$$

$$y = c*y'$$

```
extended_gcd( A, B ) {  
    r1 = A; x1 = 1; y1 = 0;  
    r2 = B; x2 = 0; y2 = 1;  
    while( r2 != 0 ) {  
        q = r1 / r2;  
        r3 = r1 - q*r2;  
        x3 = x1 - q*x2;  
        y3 = y1 - q*y2;  
        r1 = r2; x1 = x2; y1 = y2;  
        r2 = r3; x2 = x3; y2 = y3;  
    }  
    // vrijedi: x1*A + y1*B == r1  
}
```

4. Matrice – u nastavku je dana generička implementacija kvadratne matrice i neke metode koje implementiraju razne operacije s matricama.

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

template<typename T> class matrix {
private:
    int n;
    vector< vector<T> > a;
public:
    const vector<T>& operator[] (int index) const {return a[index];}

    vector<T>& operator[] (int index) {return a[index];}

    int size() const {return a.size();}

    matrix operator+ (const matrix &rhs) const {
        matrix sol(n);
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                sol[i][j] = (*this)[i][j] + rhs[i][j];
        return sol;
    }
}
```

```

matrix operator* (const matrix &rhs) const {
    matrix sol(n);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                sol[i][j] += (*this)[i][k] * rhs[k][j];
    return sol;
}

```

```

matrix operator^ (const long long pot) const {
    matrix sol(n);
    for (int i = 0; i < n; ++i)
        sol[i][i] = 1;
    for (long long x = (1LL<<62); x; x>>=1)
        sol = sol * (x & pot ? sol * (*this) : sol);
    return sol;
}

```

```

void resize(int size) {
    a.resize(n = size);
    for (int i = 0; i < n; a[i++].resize(n));
}

```

```

matrix(int size) {resize(size);}

```

```

matrix() {}
};

```

5. Potenciranje

Potenciranje broja b na potenciju p možemo najjednostavnije izvesti tako da promatramo potenciranje kao uzastopno množenje. Složenost ovakvog potenciranja jest $O(p)$ gdje je p potencija. Ako nam uvjeti zadatka dopuštaju, možemo koristiti takvo potenciranje.

```
int power(int b, int p) {
    int sol = 1;
    for (int j = 0; j < p; ++j)
        sol *= b;
    return sol;
}
```

Na sličan način, moguće je napisati i nešto ljepšu rekurzivnu funkciju uz istu vremensku složenost i lošiju memorijsku složenost (rekurzivni stog).

```
int power(int b, int p) {
    return p ? b * power(b, p - 1) : 1;
}
```

U gornja dva primjera smo iskoristili svojstvo da je $a^x == a * a^{(x-1)}$. U oba primjera složenost izvršavanja jest $O(p)$. Na sreću, postoji puno brži algoritam kojeg zovemo logaritamsko potenciranje. U njemu se koristimo relacijama:

- (1) $a^{(2x)} == (a^x)^2$
- (2) $a^x == a * a^{(x-1)}$

Rekurzija je jednostavna. Ako je potencija parna, iskoristi relaciju (1), a ako je potencija neparna iskoristi relaciju (2).

```
int power(int b, int p) {
    if (!p) return 1;
    int t = power(b, p / 2);
    return p % 2 ? b * t * t : t * t;
}
```

Na kraju, zanimljivo je vidjeti kako se ova rekurzija može implementirati bez rekurzivnih poziva. Ideja je u tome da rastavimo potenciju p na binarne znamenke, te u šetnji s lijeva na desno radimo dvije operacije ovisno o vrijednosti trenutnog bita:

- (trenutni bit je 0) kvadriraj rješenje
- (trenutni bit je 1) kvadriraj rješenje i pomnoži ga s bazom

```

int power(int b, int p) {
    int sol = 1;
    for (int x = (1<<30); x; x>>=1)
        sol = sol * sol * (x & p ? b : 1);
    return sol;
}

```

6. Faktorizacija

Važna tema u teoriji brojeva jest dakako faktorizacija. Postoji puno algoritama za faktorizaciju, no ovdje ćemo navesti dva. Najlošiji, te osrednji. Najlošiji, ili takozvani brute-force algoritam pokušava faktorizirati broj n tako da iterira svaki prirodni broj od 2 do n te razmatra djeljivost s .

Ukoliko je trenutni broj djeljiv sa n , ispiši ga kao faktor te podijeli n s njime. Mana ovog rješenja jest sporost.

```

void factor(int n) {
    int p;
    while (n > 1) {
        for (p = 2; n % p; ++p);
        for (; n % p == 0; n /= p)
            printf("%d ", p);
    }
}

```

Bolji način jest da napravimo algoritam sličan Eratostenovom situ koji će preprocesati vrijednost najmanjeg prostog faktora za svaki prirodan broj manji od neke vrijednosti (ovisno o tome koliko memorije imamo na raspolaganju).

Implementacija pred-algoritma ide ovako:

```
int nd[maxn];

void sito(int n) {
    for (int j = 2; j <= n; ++j)
        if (!nd[j])
            for (int k = j; k <= n; k += j)
                nd[k] = nd[k] ? nd[k]:j;
}
```

Nakon sto imamo popunjen niz nd (sto je skracenica od najmanji djelitelj), funkciju za faktorizaciju je lako napisati:

```
void factor(int n) {
    for (;n > 1; n /= nd[n])
        printf("%d ", nd[n]);
}
```

Mnogi algoritmi za faktorizaciju koriste heurističke metode za određivanje je li broj prost ili složen. Postoji i brži algoritam za faktorizaciju velikih brojeva smišljen 1996., no za kvantna računala.

Linearne diferencijske jednačbe

- **Fibonaccijev niz:**

- $F[0] = 1$

- $F[1] = 1$

- $F[k] = F[k-1] + F[k-2]$

- ▶ **Cilj je pronaći matricu koja će vektor**

$$\begin{bmatrix} F[k-2] \\ F[k-1] \end{bmatrix}$$

transformirati u vektor

$$\begin{bmatrix} F[k-1] \\ F[k] \end{bmatrix}$$

Linearne diferencijske jednadžbe

- **Fibonaccijev niz:**

- $F[0] = 1$

- $F[1] = 1$

- $F[k] = F[k-1] + F[k-2]$

$$\begin{bmatrix} F[k-2] \\ F[k-1] \end{bmatrix}$$

$$\begin{bmatrix} ? & ? \\ ? & ? \end{bmatrix} \begin{bmatrix} F[k-1] \\ F[k] \end{bmatrix}$$

Linearne diferencijske jednadžbe

- **Fibonaccijev niz:**

- $F[0] = 1$

- $F[1] = 1$

- $F[k] = F[k-1] + F[k-2]$

$$\begin{bmatrix} F[k-2] \\ F[k-1] \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} F[k-1] \\ F[k] \end{bmatrix}$$

Linearne diferencijske jednačbe

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} F[0] \\ F[1] \end{bmatrix} = \begin{bmatrix} F[1] \\ F[2] \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} F[0] \\ F[1] \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} F[1] \\ F[2] \end{bmatrix} = \begin{bmatrix} F[2] \\ F[3] \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^k \times \begin{bmatrix} F[0] \\ F[1] \end{bmatrix} = \begin{bmatrix} F[k] \\ F[k+1] \end{bmatrix}$$

Linearne diferencijske jednačbe

- **Primjer 1:**

- $F[0] = 0$

- $F[1] = 0$

- $F[k] = 2 \cdot F[k-1] - F[k-2] + 5$

$$\begin{bmatrix} F[k-2] \\ F[k-1] \end{bmatrix}$$

$$\begin{bmatrix} ? & ? \\ ? & ? \end{bmatrix} \begin{bmatrix} F[k-1] \\ F[k] \end{bmatrix}$$

Trebamo u stanje dodati konstantu 1!

Linearne diferencijske jednačbe

- **Primjer 1:**

- $F[0] = 0$

- $F[1] = 0$

- $F[k] = 2 \cdot F[k-1] - F[k-2] + 5$

$$\begin{bmatrix} F[k-2] \\ F[k-1] \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix} \begin{bmatrix} F[k-1] \\ F[k] \\ 1 \end{bmatrix}$$

Linearne diferencijske jednačbe

- **Primjer 1:**

- $F[0] = 0$

- $F[1] = 0$

- $F[k] = 2 \cdot F[k-1] - F[k-2] + 5$

$$\begin{bmatrix} F[k-2] \\ F[k-1] \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 0 \\ -1 & 2 & 5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} F[k-1] \\ F[k] \\ 1 \end{bmatrix}$$

Linearne diferencijske jednačbe

- **Primjer 2:**
 - $F[0] = 0$
 - $F[1] = 0$
 - $F[k] = 2 \cdot F[k-1] - F[k-2] + 5$
 - Izračunajte $F[0] + F[1] + F[2] + \dots + F[n]$
- **Rješenje: niz S pamti sumu**
 - $S[0] = 0$
 - $S[k] = S[k-1] + F[k]$
- $S[k] = S[k-1] + 2 \cdot F[k-1] - F[k-2] + 5$

Linearne diferencijske jednačbe

- **Primjer 2:**

- $F[0] = 0$

- $F[1] = 0$

- $F[k] = 2 \cdot F[k-1] - F[k-2] + 5$

- $S[k] = S[k-1] + 2 \cdot F[k-1] - F[k-2] + 5$

- **Vektor stanja:**

$$\begin{bmatrix} F[k-2] \\ F[k-1] \\ S[k-1] \\ 1 \end{bmatrix}$$

Linearne diferencijske jednačbe

- **Primjer 2:**

- $F[0] = 0$

- $F[1] = 0$

- $F[k] = 2 \cdot F[k-1] - F[k-2] + 5$

- $S[k] = S[k-1] + 2 \cdot F[k-1] - F[k-2] + 5$

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 2 & 0 & 5 \\ -1 & 2 & 1 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} F[k-2] \\ F[k-1] \\ S[k-1] \\ 1 \\ F[k-1] \\ F[k] \\ S[k] \\ 1 \end{bmatrix}$$