VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Implementace kódů proměnné délky pro celá čísla

Implementation of Variable-length Codes for Integer Numbers

2010

Radek Michálek

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava*.

V Ostravě 7. května 2010 .....

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2010 .....

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli, protože bez nich by tato práce nevznikla.

Abstrakt

Cílem práce bylo seznámit se s několika typy kódování proměnných délek pro celá čísla, provést jejich implementaci z pohledu možného nasazení v kompresním frameworku a porovnat je z pohledu jejich efektivity. Zvoleny byly Fibonacciho kódy, Zobecněné Fibonacciho kódy, Goldbachovy kódy, Aditivní kódy, Golombovy a Riceovy kódy.

Při hodnocení efektivity kódování jsem se zaměřil na to, zda a za jakých podmínek je dané kódování přínosem v porovnání s běžnou třiceti-dvoubitovou znaménkovou binární reprezentací.

Implementace byla s výjimkou Aditivních kódů a Goldbachova G0 kódu provedena pro rozsah vstupních hodnot $< 1, 2147483647 >$. Po konzultaci s vedoucím práce bylo od implementace Aditivních kódů upuštěno a v případě Goldbachova G0 kódu byla implementace omezena na maximální délku kódu 120 000 bitů.

Porovnání efektivity bylo provedeno na poskytnuté sadě testovacích souborů, přičemž v případě Goldbachova kódování G0 byl rozsah testování omezen.

Klíčová slova: Fibonacciho kódy, Zobecněné Fibonacciho kódy, Goldbachovy kódy, Aditivní kódy, Golombovy a Riceovy kódy

Abstract

The goal of the works was to acquaint with several types of encoding variable lenght for integral numbers, to analyze their implementation in terms of potencional use in the compression framework and to compare their effectiveness. I chose the Fibonacci code, the Generalized Fibonacci code, the Goldbach codes, the Additive codes, the Golomb and the Rice code. When evaluating the encoding efficiency, I focused on wether and under what conditions is the encoding useful in comperision with regular thirty-two sign binary representation.

The implementation was made for a range of input value $< 1, 2147483647 >$ expect the Additive and Goldbach codes G0. The implementation of the Additive codes was canceled and the implementation of the Goldbach codes G0 was limited to a maximum code leght 120 000 bites, after the consultation with the supervisor.

The comparing of the effectiveness was made on the provided test files. In case of Goldbach codes G0 the extent of testing was limited.

Keywords: Fibonacci code, Generalized Fibonacci code, Goldbach Codes, Additive Codes, Golomb and Rice code

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Problém reprezentace celých čísel je v informatice řešen většinou pomocí znaménkových či neznaménkových binárních čísel pevné délky. Nevýhodou tohoto řešení je nezávislost délky výsledného kódu na hodnotě kódovaného čísla. Účelem této práce je zaměřit se na kódování, u nichž je velikost binární reprezentace závislá na velikosti čísla a případně ještě parametru.

Ačkoli je princip některých těchto kódování znám již mnohdy i více než sto let, dostává se jim většího významu teprve se vzrůstajícím výkonem moderních počítačů. Snahou je minimalizovat množství ukládaných nebo přenášených dat.

2 Principy kódování

Jak vidíme v tabulce 1 mají jednotlivá kódování k uchování hodnoty čísla n dosti různorodé přístupy.

2.1 Fibonacciho kódovaní

Fibonacciho kódování staví na základech položených italským učencem Leonardem Fibonaccim z Pizy, žijícím na přelomu 12. a 13. století a práci belgického lékaře a matematika Eduarda Zeckendorfa.

Fibonacci použil dnes po něm pojmenovanou nekonečnou číselnou řadu k popisu růstu králičí populace za zidealizovaných podmínek. Řada byla definována takto :

$$m = 2$$

$$F_i^{(m)} = F_{i-1}^{(m)} + F_{i-2}^{(m)} + ... + F_{i-m}^{(m)}, fori \geq 1$$

$$where\ F_{-m+1}^{(m)} = F_{-m+2}^{(m)} = ... = F_{-2}^{(m)} = 0, F_{-1}^{(m)} = F_0^{(m)} = 1.$$

Eduard Zeckendorf objevil to, že jakékoli kladné celé číslo je možné zapsat jako součet 1 až n různých, spolu nesousedících čísel fibonacciho posloupnosti. Tento matematický jev je dnes znám jako Zeckendorfův theorem a může být využit k vytvoření číselné soustavy. Na rozdíl od jiných soustav, kde přítomnost symbolu x na pozici i znamená, že číslo obsahuje $x \times k^i$, kde k je základ číselné soustavy. V této soustavě znamená přítomnost symbolu x na pozici i, že číslo obsahuje $x \times F_i$, kde F_i je i-tý prvek nekonečné posloupnosti $1, 2, 3, 5, ...$ a $x \in < 0, 1 >$. Čísla této soustavy budu nadále nazývat Fibonacciho čísla řádu dva a budu je značit $Z_i^{(2)}$.

Fibonacciho kódování využívá toho, že fibonacciho číslo řádu dvě pro $n \geq 1$ začíná vždy jedničkou a nikde se v něm nevyskytují dvě jedničky vedle sebe. Pokud ho tedy zapíšeme pozpátku a připojíme za něj jedničku, získáme v datovém streamu jasně rozlišitelný sufix.

2.2 Zobecněné (Generalizované) Fibonacciho kódovaní

Zobecněné Fibonacciho kódy byly poprvé prezentovány profesory Albertem Apostolicem a Avierzi Frankelem, kteří ukázali jejich snadnou kódovatelnost a dekódovatelnost.

Zobecněné Fibonacciho kódy řádu m sestávají podobně jako Fibonacciho kódy z číselné reprezentace a sufixu, s výjimkou případu, kdy kódované číslo $n = 1$, kdy $F^{(m)}(n) = 1_m$ nebo $n = 2$, pak se kód rovná pouze sufixu, tedy $F^{(m)}(n) = 01_m$. Číselná reprezentace je tvořena uspořádanou řadou jedinečných 1 až n bitových kombinací neobsahujících m jedničky za sebou, seřazených podle délky. Přičemž číslu n odpovídá prvek řady na pozici $n - 2$.

Příklad 2.1
$m = 3; n = 16; 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 0000, ... => F(n) = 0000 =>$
$F^{(3)} = 00000111.$ ■

n	Fibonacciho kód	Délka	Gen. Fib. kód m = 3	Délka	G0 kód	Délka	Golomb. kód m = 14	Délka
1	11	2	111	3	11	2	0001	4
2	011	3	0111	4	101	3	00100	5
3	0011	4	00111	5	011	3	00101	5
4	1011	4	10111	5	1001	4	00110	5
5	00011	5	000111	6	0101	4	00111	5
6	10011	5	010111	6	0011	4	01000	5
7	01011	5	100111	6	00101	5	01001	5
8	000011	6	110111	6	010001	6	01010	5
9	100011	6	0000111	7	00011	5	01011	5
10	010011	6	0010111	7	0010001	7	01100	5
11	001011	6	0100111	7	000101	6	01101	5
12	101011	6	0110111	7	000011	6	01110	5
⋮	⋮		⋮		⋮		⋮	

Tabulka 1: Fibonacciho, Generalizované Fibonacciho, G0 Goldbachovo a Golombovo kódování

Pro účely této práce mi bylo konzultantem práce zadáno vypracovat Zobecněné Fibonacciho kódování pro m rovno 3, 4 a 5. V počátcích práce na tomto kódování jsem se snažil k získání číselné reprezentace využít "brutal force" algoritmus s narůstajícím počtem kódovaných čísel a jejich rostoucí hodnotou se však použití takto neefektivního algoritmu stalo neúnosným a po seznámení s prací pánů Jiřího Weldera a Michala Krátkého[1], jsem přistoupil k použití Fast Coding a Decoding Algoritmu.

Fast Coding Algoritmus pro $n = 1$ a $n = 2$ přiřazuje $F^{(m)}(1) = 1_m$ a $F^{(m)}(2) = 01_m$. Pro $n > 2$ hledáme k, pro které platí $S_{k-2}^{(m)} < n \leq S_{k-1}^{(m)}$. Nalezené k pak použijeme k výpočtu $Q = n - S_{k-2} - 1$. $F^{(m)}(Q)$ následně doplníme zprava nulami na k míst, čímž získáme číselnou reprezentaci, kterou doplníme sufixem 01_m.

$$S_n^{(m)} = \begin{cases} 0, pro & n < -1 \quad . \\ \sum_{i=-1}^{n} F_i^{(m)}, pro & n \geq -1 \quad . \end{cases}$$

2.3 Aditivní kódy

Aditivní kódy využívají základní vlastnosti aditivní sekvence, kdy jakékoli kladné celé číslo lze zapsat jako součet dvou čísel aditivní sekvence. V příkladu 2.2 a 2.3 je znázorněno odvozování aditivní řady $(0, 1, 2, ..)$ a $(0, 1, 2, ..)$. Pokud nemůžeme číslo zapsat jako součet dvou čísel sekvence přidáme ho do sekvence.

Příklad 2.2
Základ sekvence $= (0, 1, 2)$
$0 = 0 + 0$
$1 = 0 + 1$
$2 = 1 + 1$ nebo $0 + 2$
$3 = 1 + 2$
$4 = 2 + 2$
$5 = 0 + 5 =>$ přidáme do sekvence
$6 = 1 + 5$ nebo $2 + 4$
$7 = 2 + 8$
$5 = 0 + 8 =>$ přidáme do sekvence
$\vdots$ ■

Příklad 2.3
Základ sekvence $= (0, 1, 3)$
$0 = 0 + 0$
$1 = 0 + 1$
$2 = 1 + 1$
$3 = 0 + 3$
$4 = 1 + 3$
$5 = 0 + 5 =>$ přidáme do sekvence
$6 = 1 + 5$ nebo $2 + 4$
$7 = 0 + 7 =>$ přidáme do sekvence

$5 = 1 + 7$

$\vdots$ ■

Aditivní kódování reprezentuje tento součet, kdy $n = a_i + a_j$ a $a_i \leq a_j$, pomocí dvou binárních čísel z nichž obě jsou doplněny o prefix, a první je rovno i , druhé $j - (i - 1)$, přičemž i reprezentuje index menšího z čísel sekvence a $j - (i - 1)$ je ofsetem indexu pro a_j. Prefix pro x bitové číslo je tvořen $x - 1$ nulami.

Vzhledem k tomu, že indexy prvků sekvence potřebných k zakódování n budou vždy menší než n, přičemž druhý z indexů je zadán ofsetem. Lze usuzovat na poměrně dobrou efektivitu kódování v určité omezené oblasti. Nevýhodou je narůstající délka prefixů pro obě části kódu, a to že z délky kódu pro n nemůžeme odhadnout délku kódu pro $n + 1$.

2.4 Goldbachovo kódování

U Goldbachova G0 kódování vycházíme z předpokladu, že pro každé celé kladné n platí $2(n + 3) = P_i + P_j$, přičemž P_i a P_j jsou prvočísla, pro které platí $P_i < P_j$ a současně $P_i \geq 3$.

Výraznou nevýhodou Goldbachova G0 kódování je prudký nárůst délky kódu s rostoucím n. Problém může také představovat to, že délka kódu pro dvě sousedící n se může značně lišit. První z těchto neduhů vedl k omezení implementace a nakonec i testování. Druhý byl příčinou toho, proč došlo k omezení implementace na základě délky výstupního kódu a ne k pouhému omezení rozsahu vstupního intervalu.

2.5 Golombovy a Riceovy kódy

Riceovy kódy jsou pouze specifickým případem Golombových kódů, kdy je parametr m roven mocnině čísla 2.

Golombovy kódy sestávají z prefixu a binárního čísla. Prefix je tvořen řadou j jedniček následovaných nulou, přičemž $n = (j * m) + q$, kde j je výsledkem celočíselného dělení n/m a q je zbytkem po tom to dělení, uloženým v binárním čísle následujícím po prefixu.

V případě, že je parametr m mocninou dvojky, je délka binárního kódu pro zbytek konstantní a rovná se $C = \lceil \log_2 m \rceil$. Pro m různé od mocnin dvojky bude pro $q < x; x = 2^C - m$ délka binárního čísla $C - 1$, pro $q \geq x$ bude rovna C.

3 Implementace

Implementace probíhala v programovacím jazyce C++. Při implementaci byly využité třídy a hlavičkové soubory cCoder, cBitImputStream a cBitOutputStream vytvořené vedoucím práce a uvolněné pod GPL licencí.

Abstraktní třída cCoder představuje společné rozhraní pro třídy kodérů/dekodérů zadaných kódování a poskytuje funkci pro výpočet $\log_2$. Všechny mnou vytvořené kodéry/dekodéry obsahují funkci int encode(int n) pro kódování, int decode() pro dekódování, a int Length(int n) pro zjištění délky kódu pro n.

3.1 Implementace Fibonacciho kódování

Většinou nejjednodušší funkci všech kodérů představoval výpočet délky výsledného kódu. V případě Fibonacciho kódování postačí znát index největšího prvku Fibonacciho posloupnosti obsaženého v čísle n. Cyklus se ukončí ve chvíli, kdy je nalezeno číslo rovné n nebo první větší. Na základě toho pak přičteme pro číslo rovno n dvě (jedna za sufix a jedna jako kompenzace toho, že prvky v poli jsou číslovány od nuly) a pro číslo větší jedna (pouze za sufix, číslování pole je už kompenzováno nalezením většího čísla).

```
int cFibonaciCoder :: Length(int n) {
    int pocitadlo = 0;
    while(this->rada[pocitadlo] < n)
    {
        pocitadlo++;
    }
    if (this->rada[pocitadlo] == n)
    {
        return pocitadlo + 2;
    }
    else
    {
        return pocitadlo + 1;
    }

}
```

Výpis 1: funkce int cFibonaciCoder::Length(int n)

Problematiku kódování pomocí Fibonacciho kódu jsem si ve třídě cFibonaciCoder rozdělil na výpočet Fibonacciho čísla řádu dva a zápis do souboru s doplněním sufixu. Výpočet Fibonacciho čísla řeším pomocí rekurzivní funkce std::string fibS(int n). Při výpočtu Fibonacciho čísla řádu dva hledám největší číslo Fibonacciho posloupnosti, které je menší a nebo rovno n. Pokud je nalezené číslo rovno n, vrátím jeho pozpátku do řetězce zapsanou reprezentaci ve formě Fibonacciho čísla řádu dvě, jestliže je nalezené číslo menší než n, použiji rekurzy k získání Fibonacciho čísla řádu dvě pro rozdíl n a nalezeného čísla. Provedu "bitový součet"(čísla jsou reprezentována formou řetězce a tak bitový součet v pravém slova smyslu nelze použít) fibonacciho reprezentace řádu dvě nalezeného čísla a výsledku rekurze.

```
std :: string  cFibonaciCoder ::fibS(int  n)
{
    int  pocitadlo  = 0;
    std :: string  res;
    res.reserve(50);
    while(this->rada[pocitadlo] < n)
    {
        pocitadlo++;
    }
    if (this->rada[pocitadlo] == n)
    {
        for(int  i  = 0;  i  < pocitadlo;i++)
        {
            res.append("0");
        }
        res.append("1");
    }
    else
    {
        pocitadlo  = pocitadlo  - 1;
        std :: string  pom;
        pom.reserve(50);
        pom = this->fibS(n - this->rada[pocitadlo]);
        for(int  i  = 0;  i  < pocitadlo;i++)
        {
            if (  i  < pom.length() )
            {
                if (pom[i] == '1')
                {
                    res.append("1");
                }
                else
                {
                    res.append("0");
                }
            }
            else
            {
                res.append("0");
            }
        }
        res.append("1");
    }
    return res;
}
```

Výpis 2: funkce int cFibonaciCoder::fibS(int n)

Volání funkce fibS, zápis do bitového streamu a doplnění sufixu realizuje triviálním způsobem funkce int Encode(int n).

```
int  cFibonaciCoder :: Encode(int n) {
    std :: string  res;
    res.reserve(50);
```

```
    res = this->fibS(n);
    if (symbol == 1)
    {
        m_OutputStream->WriteBit(1);
        m_OutputStream->WriteBit(1);
    }
    else
    {
    for(int i = 0; i <res.length(); i++ )
    {
        if (res[i] == '1')
        {
            m_OutputStream->WriteBit(1);
        }
        else
        {
            m_OutputStream->WriteBit(0);
        }
    }
    m_OutputStream->WriteBit(1);
    }
    return CODER_OK;
}
```

Výpis 3: funkce int cFibonaciCoder::Encode(int n)

Při dekódování postupně načítám jednotlivé bity, přičemž inkrementuji počítadlo. Pokud je načtený bit jednička a jednička mu nepředcházela, přičtu do proměnné res s výsledkem číslo fibonacciho posloupnosti, uložené v poli rada na souřadnicích počítadla. Pokud narazím na druhou jedničku v řadě, ukončím načítání a vrátím obsah proměnné res.

```
int cFibonaciCoder :: Decode() {
    int pom1 = 0;
    int pom2;
    int res=0;
    int pocitadlo=-1 ;
    while(true){
        pom2 = m_InputStream->ReadBit();
        if ((pom1 == 1) && (pom2 == 1))
        {
            break;
        }
        else
        {
            pocitadlo++;
            pom1=pom2;
            if (pom1 == 1){
                res=res + (int)this->rada[pocitadlo];
            }
        }
    }
    return res;
}
```

Výpis 4: funkce int cFibonaciCoder::Decode(int n)

3.2 Implementace Generalizovaného Fibonacciho kódování

Implementace funkce pro zjišťování délky kódu, stejně jako funkce pro kódování a dekódování, jsou postaveny na principech publikovaných v [1]. Vycházím z toho, že délka kódu je rovna $m + k$.

```
int cGenFibonaciCoder :: Length(int symbol) {
    if (symbol == 1)
    {
        return this->gm;
    }
    if (symbol == 2)
    {
        return this->gm+1;
    }
    for(int i = 1 ; i < 35; i++)
    {
        if ((this->suma[i-1]<symbol))
        {
            if (symbol <= this->suma[i])
            {
                return i+1+this->gm;//souradnice vpoli =k-1; delka kodu = k + m
            }
        }
    }

}
```

Výpis 5: funkce int cGenFibonaciCoder :: Length(int symbol)

Implementace kódování odpovídá Fast Coding Algoritmu popsanému v kapitole 2.2, přičemž jsem činnosti spojené s kódováním opět rozdělil do dvou celků. Pro výpočet Fibonacciho čísel řádu 3, 4 a 5 využívám identickou funkci fibS, která má ovšem k dispozici v paměti odlišné posloupnosti. Ve funkci int encode(int n) je pak řešeno doplnění případných nul zprava a sufixu.

```
int cGenFibonaciCoder :: Encode(int symbol) {

    if (symbol == 1)
    {
        for(int i =0; i < this->gm;i++)
        {
        m_OutputStream->WriteBit(1);
        }
    }
    else
    {
    if (symbol == 2)
```

```
    {
        m_OutputStream->WriteBit(0);
        for(int i =0; i < this->gm;i++)
        {
        m_OutputStream->WriteBit(1);
        }

    }
    else
    {
    std::string q;
    q.reserve(100);
    int k;
    for(int i = 1 ; i < 35; i++)
    {
        if ((this->suma[i-1]<symbol))
        {
            if (symbol <= this->suma[i])
            {
                q = fibS(symbol - this->suma[i- 1] -1);
                k = i+1;
                break;
            }
        }
    }
    for(int i = 0; i < q.length();i++)
    {
        if (q[i] == '1')
        {
            m_OutputStream->WriteBit(1);
        }
        else
        {
            m_OutputStream->WriteBit(0);
        }
    }
    for(int i = 0; i< (k -1 - q.length());i++)
    {
            m_OutputStream->WriteBit(0);
    }
    m_OutputStream->WriteBit(0);
        for(int i =0; i < this->gm;i++)
        {
        m_OutputStream->WriteBit(1);
        }
    }
    }
    return CODER_OK;
}
```

Výpis 6: funkce int cGenFibonaciCoder :: Encode(int symbol)

3.3 Implementace Goldbachova G0 kódu

Vzhledem k výrazné neefektivitě tohoto kódování, jsme omezili implementaci na vstupní hodnoty n, pro něž délka výstupního kódu nepřekročí 120 000 bitů. Jistý problém pro implementaci představuje nejednoznačnost kódu, kdy $2(n+3)$ můžeme zapsat pomocí více různých součtů dvou nestejných prvočísel. Ve své implementaci se snažím najít vždy dvě prvočísla, jejichž rozdíl indexů v řadě je co nejmenší. Toto opatření mi zaručuje, že ze všech možných vyberu vždy ten nejkratší goldbachův G0 kód.

Na rozdíl od triviálního algoritmu pro zjišťování délky Fibonacciho kódování 3.1 je algoritmus mnohem složitější a výpočetně náročnější. Pro realizaci algoritmu jsem kromě funkce int Length implementoval funkci int hledej, která v poli 120 000 prvočísel vyhledává půlením intervalu a vrací index nalezeného čísla. Používám ji také ke kontrole, zda je číslo prvočíslem. V případě, že číslo není prvočíslem v daném intervalu $< 3, 1583273 >$, vrací -1 .

```
int cGoldbachG0Coder::hledej(int cislo,int dolnimez,int hornimez)
{

    if (dolnimez>hornimez)
    {
        return -1;
    }
    int stred = (dolnimez + hornimez)/2;

    if (prvocisla[stred] == cislo)
    {
        return stred+ 1;
    }
    if ( cislo < prvocisla[stred])
    {
        return hledej( cislo ,dolnimez, stred - 1);
    }
    else
    {
        return hledej( cislo ,stred + 1, hornimez);
    }

}
```

Výpis 7: funkce int cGoldbachG0Coder::hledej(int cislo, int dolnimez, int hornimez)

Funkce int Length prochází postupně všechny součty dvou celých čísel, které se rovnají $2(n+3)$ a hledá mezi nimi součet dvou prvočísel s co nejmenším odstupem indexů. Protože z principu kódování víme, že délka kódu je rovna indexu většího z čísel součtu, vracíme jeho index.

```
int cGoldbachG0Coder::Length(int cislo)
{
    int index1 = 0;
    int index2 = 0;
    int pom1;
    int pom2;
```

```
    for(int i = 1;i <= (2*(cislo + 3));i++)
    {
        pom1 = hledej(i,0,120000);
        pom2 = hledej((2*(cislo + 3))-i,0,120000);
        if ((pom1 != -1)&& (pom2 != -1))
        {
            if (pom1 < pom2)
            {
                index1 = pom1;
                index2 = pom2;
            }
        }
    }
    return index2;
}
```

Výpis 8: funkce int cGoldbachG0Coder::Length(int cislo)

Při implementaci funkce int Encode(int symbol) jsem při hledání indexů vycházel z funkce Length a doplnil zápis do streamu.

```
int cGoldbachG0Coder::Encode(int cislo)
int index1 = 0;
    int index2 = 0;
    int pom1;
    int pom2;
    for(int i = 1;i <= (2*(cislo + 3));i++)
    {
        pom1 = hledej(i,0,120000);
        pom2 = hledej((2*(cislo + 3))-i,0,120000);
        if ((pom1 != -1)&& (pom2 != -1))
        {
            if (pom1 < pom2)
            {
                index1 = pom1;
                index2 = pom2;
            }
        }
    }
    for(int j = 0; j<index2; j++)
    {
        if ((j+1) == index1)
        {
            // printf ("1") ;
            m_OutputStream->WriteBit(1);
        }
        else
        {
            if ((j+1) == index2)
            {
                // printf ("1") ;
                m_OutputStream->WriteBit(1);
            }
            else
```

```
            {
                // printf ("0") ;
                m_OutputStream->WriteBit(0);
            }
        }
    }
    return CODER_OK;
}
```

Výpis 9: funkce int cGoldbachG0Coder::Encode(int cislo)

Při dekódování Goldbachova G0 kódu postupujeme tak, že postupně ze streamu načítáme jednotlivé bity a pokud narazíme na jedničku, inkrementujeme počítadlo (jedniček) a do proměnné uchovávající výsledek přičteme prvočíslo jehož index odpovídá pořadí načteného bitu s jedničkou. Pokud počítadlo (jedniček) signalizuje načtení druhé jedničky, ukončíme načítání. Poté výsledek vydělíme dvěma a odečteme tři.

```
int cGoldbachG0Coder::Decode()
{
    int pocitadlo = 0;
    int jednicek = 0;
    int vysledek = 0;

    while(jednicek <2)
    {
        if (m_InputStream->ReadBit() == 1)
        {
            vysledek = vysledek + prvocisla[pocitadlo];
            jednicek++;
        }
        pocitadlo++;
    }
    return (vysledek / 2)-3;
}
```

Výpis 10: funkce int cGoldbachG0Coder::Decode(int cislo)

3.4 Implementace Golombova a Riceova kódu

Jak můžeme vidět ve zdrojovém kódu, spočívá zjišťování délky Golombova kódu ve výpočtu q a a, kdy pro $a < x$ je délka výsledného kódu $q + 1 + c - 1$ a pro $a \geq x$ $q + 1 + c$. U Riceova kódu může nastat pouze druhý případ.

```
int cGolombCoder::Length(int symbol)
{
    int res = 0;
    if ((symbol / this->gm) == 0)
    {
        res++;
    }
    else
```

```
    {
        res = (symbol / this->gm) + 1;
    }
    if ((symbol % this->gm) < this->gx)
    {
        res = res + (this->gc - 1);
    }
    else
    {
        res = res + this->gc;
    }
    return res;
}
```

Výpis 11: funkce int cGolombCoder::Length(int symbol)

Při kódování zjišťuji výsledek celočíselného dělení q a zbytek po celočíselném dělení r. Pomocí cyklu zapíši do souboru q jedniček následovaných nulou. Na základě velikosti r vyhodnotím, zda má A délku C nebo $C - 1$. Pomocí funkce rTostring získám řetězec reprezentující A, neboli r jako binární číslo délky C nebo $C - 1$. Provedu zápis A do streamu.

```
int cGolombCoder::Encode(int symbol)
{
    int r = symbol % this->gm;
    int q = symbol / this->gm;
    std::string a;
    a.reserve(40);
    for(int i = 0; i < q;i++ )
    {
        m_OutputStream->WriteBit(1);
        // printf ("1") ;
    }
    m_OutputStream->WriteBit(0);
    // printf ("0") ;
    if (r < this->gx)
    {
        a = this->rTostring(r,this->gc - 1);
    }
    else
    {
        a = this->rTostring(r + this->gx,this->gc );
    }
    for(int i = 0; i < a.length();i++)
    {
        if (a[i] == '1')
        {
            m_OutputStream->WriteBit(1);
            printf ("1");
        }
        else
        {
            m_OutputStream->WriteBit(0);
```

```
            printf ("0");
        }
    }

    return CODER_OK;
}
```

Výpis 12: funkce int cGolombCoder::Encode(int symbol)

```
std :: string  cGolombCoder::rTostring(int r,int  bits )
{
    std :: string  res;
    res.reserve(40);
    for(int  i  = 0  ;  i<bits;  i++)
    {
        if (((r  >> (bits -i -1)) & 1 ))
        {
            res.append("1");
        }
        else
        {
            res.append("0");
        }
    }
    return res;
}
```

Výpis 13: funkce int cGolombCoder::rTostring(int r, int bits)

Při dekódování Golombova kódu načítám ze streamu tak dlouho, dokud nenarazím na nulu, přičemž inkrementuji počítadlo. Počítadlo je nyní rovno q. Načtu ze streamu C bitů a uložím jejich reprezentaci do řetězce. Pomocí funkce binarTodec převedu hodnotu uchovávanou řetězcem na integer. Pokud je číslo menší jak X, skončím s načítáním a vrátím výsledek výpočtu $q * m + a$. Pro čísla větší jak X načtu ještě jeden bit do řetězce a použiji znovu funkci binarTodec a vrátím výsledek $q * m + A - X$.

```
int  cGolombCoder::Decode()
{
    int  res = 0;
    int  q = 0;
    int  nul = 0;
    std :: string  a;
    a.reserve(40);


    while(nul == 0)
    {
        if (m_InputStream->ReadBit() == 1)
        {
            q++;
```

```
        }
        else
        {
            nul++;
        }
    }
    for(int i = 0; i < (this->gc - 1); i++)
    {
        if (m_InputStream->ReadBit() == 1)
        {
            a.append("1");
        }
        else
        {
            a.append("0");
        }
    }
    if (this->binarTodec(a) < this->gx)
    {
        return (this->gm * q) + this->binarTodec(a.c_str());
    }
    else
    {
        if (m_InputStream->ReadBit() == 1)
        {
            a.append("1");
        }
        else
        {
            a.append("0");
        }

        return (this->gm * q) + this->binarTodec(a.c_str()) - this->gx;
    }

}
```

Výpis 14: funkce int cGolombCoder::Decode()

```
int cGolombCoder::binarTodec(std::string bin)
{
    int res = 0;
    for(int i = 0 ; i< bin.length() ; i++)
    {
        if (bin[i]=='1')
        {
            res = res + (int)pow((double)2,(int)(bin.length() - 1 -i));
        }
    }
    return res;

}
```

Výpis 15: funkce int cGolombCoder::binarTodec(std::string bin)

4 Testování

Pro testování mi bylo poskytnuto 6 testovacích souborů. V pěti případech soubory obsahují data s uniformním rozložením v rozsahu 2^x až $2^y - 1$, kdy názvy souborů splňují konvenci data_x_y.txt (data_00_05.txt, data_00_08.txt, data_08_16.txt, data_16_24.txt, data_24_31.txt). Šestý soubor data_norm.txt obsahuje data s normálním (Gaussovým) rozložením $f(x) = \frac{1}{s*\sqrt{2*\pi}} e^{-\frac{1}{2}\left(\frac{x-m}{s}\right)^2}$, přičemž $m = 0$ a $s = 128$. Ve všech případech soubory obsahovaly milion hodnot.

Abych vyloučil jakýkoli vliv souborového a operačního systému, prováděl jsem při testování kromě kódování souboru i výpočet délky výsledného streamu v bitech, přičemž tyto vypočtené hodnoty byly použity k porovnání kódování.

V případech, kde to rozsah vstupních hodnot umožňoval, jsem provedl kromě porovnání s 32 bitovou binární reprezentací i porovnání s 8 a 16 bitovou reprezentací.

4.1 Uniformní rozložení

4.1.1 Uniformní rozložení v rozsahu 1 až 31

Malá čísla představují oblast, ve které představují testovaná kódování většinou největší přínos v porovnání s binárním kódem pevné délky 32 bitů a na rozdíl od 16 a 8 bitového kódování si zachovávají schopnost uchovat hodnotu čísla v plném rozsahu $< 1, 2147483647 >$.

Jak vidíme v grafu na obrázku 1 a v tabulce 2, je pro tento rozsah čísel za daných parametrů nejefektivnější Fibonacciho kódování, které dokázalo testovací soubor zakódovat s úsporou 79,719% v porovnání s 32-bitovým kódováním.

Dobrého výsledku by bylo možné dosáhnout i s Golombovým kódováním v případě, kdy bychom zvolili parametr m blízký horní uzávěře intervalu $< 1, 31 >$, kdy se délka kodu čísla rovná $\lceil log_2 m \rceil + 1$ pro m větší než horní uzávěra intervalu vstupních hodnot. V případě že $m = 32$ je pak délka zakódovaného souboru 6000000 bitů, což znamená 81,25% úsporu místa v porovnání s 32-bitovým kódováním.

V případě Generalizovaných Fibonacciho kódů se projevuje větší měrou délka sufixu, která je částečně kompenzována tím, že oproti Fibonacciho kódu existuje méně binárních čísel, která by obsahovala jeho sufix. Tato vlastnost se bude stále více projevovat s rostoucím rozsahem vstupních hodnot.

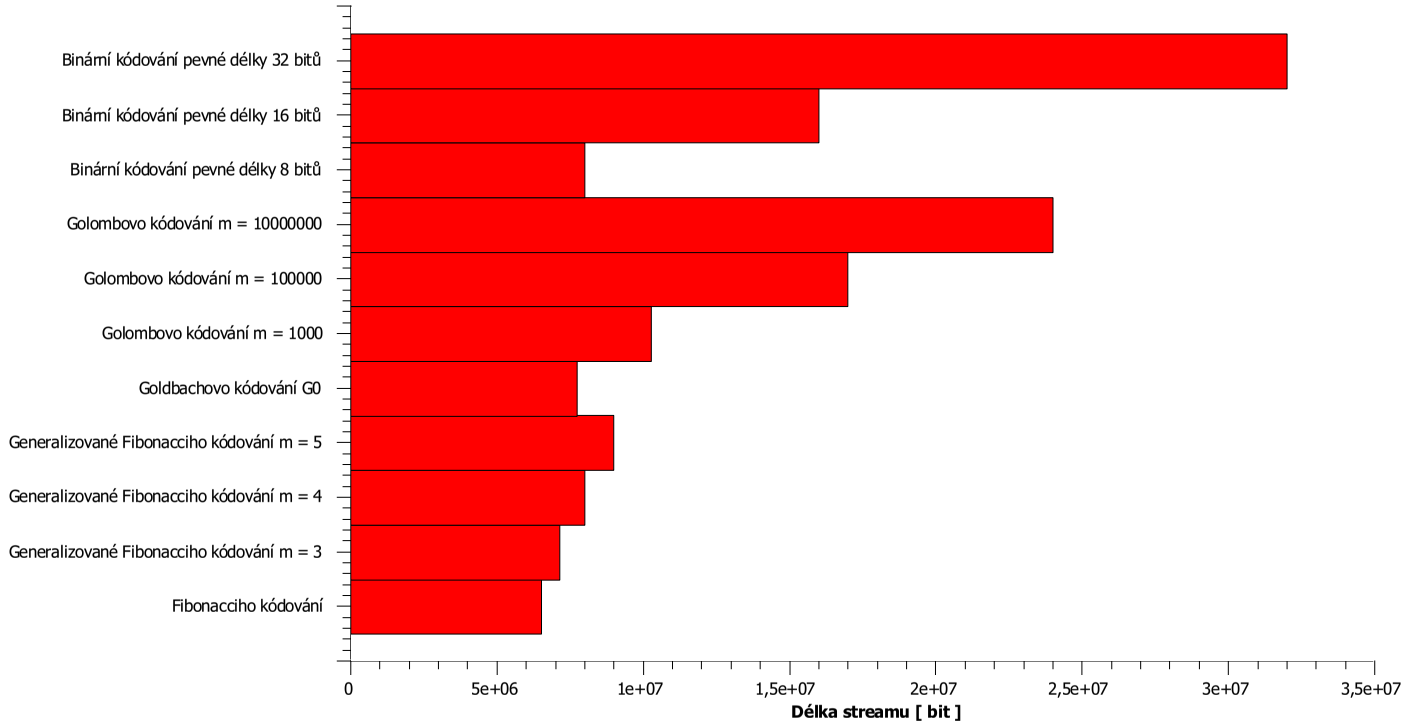
4.1.2 Uniformní rozložení v rozsahu 1 až 255

Pro rozsah vstupních hodnot 1 až 255 se etalonu v podobě 8 bitového binárního kódu pevné délky nejvíce přiblížilo Generalizované Fibonacciho kódování s parametrem $m = 3$, které vzhledem k 32 bitovému kódování uspořilo 67,194 % místa.

Při porovnání tabulek 2 a 3 si můžeme všimnout rozdílu v délkách výsledných streamu pro Golombovo kódování s parametrem $m = 1000$, ke kterému došlo v důsledku volby parametru různého od celé mocniny dvojky. Víme, že $X = 2^{\lceil \log_2 m \rceil} - m = 2^{\lceil \log_2 1000 \rceil} - 1000 = 24$, z čehož vyplývá, že pro čísla v rozsahu $< 1, 23 >$ je délka kódu 9 bitů a

Kódování	Délka streamu $[bit]$	Procentuální rozdíl vůči 32 bitovému číslu
Fibonacciho kódování	6 516 854	-79,635
Generalizované Fibonacciho kódování m = 3	7 129 929	-77,719
Generalizované Fibonacciho kódování m = 4	8 001 244	-74,996
Generalizované Fibonacciho kódování m = 5	9 001 244	-71,871
Goldbachovo kódování G0	7 742 204	-75,806
Golombovo kódování m = 1000	10 258 036	-67,944
Golombovo kódování m = 100000	17 000 000	-46,875
Golombovo kódování m = 10000000	24 000 000	-25,000
Binární kódování pevné délky 8 bitů	8 000 000	-75,000
Binární kódování pevné délky 16 bitů	16 000 000	-50,000
Binární kódování pevné délky 32 bitů	32 000 000	0,000

Tabulka 2: Délka streamů pro uniformní rozložení v rozsahu 1 až 31



Obrázek 1: Graf - Délka streamu pro uniformní rozložení v rozsahu 1 až 31

pro rozsah $< 24,999 >$ je to 10 bitů. U Golombova kódu s parametry $m = 100000$ a $m = 10000000$ tento jev ještě pozorovat nemůžeme kvůli intervalu vstupních hodnot, jehož horní uzávěra je menší než X a kód má tak konstantní délku $\lceil \log_2 m \rceil - 1 + q + 1$.

Při použití Goldbachova G0 kódu už nedochází ke kompresi, ale nárůstu délky streamu. Vzhledem k předpokládanému dalšímu výraznému nárůstu délky, nebude pro další testování využit.

4.1.3 Uniformní rozložení v rozsahu 256 až 65535

Pro rozsah vstupních hodnot 256 až 65535 je nejlepších výsledků dosaženo Golombovým kódem s parametrem $m = 100000$, naopak nejhorších výsledků pro tento rozsah, podle očekávání dosahoval Golombův kód $m = 1000$, kdy se v důsledku nárůstu q výrazně zvětšovala délka kódu (v nejhorším případě až o 65 bitů), a tak došlo k překročení délky stremu s 32 bitovým kódem. Nedochází už tedy ke kompresi, ale nárůstu délky. V případě Golombova kódu s parametrem $m = 10000000$ je vzhledem k velikosti vstupních hodnot prováděno stále kódování "pevné" délky 24 bitů.

V případě Fibonacciho kódování se stále více projevuje častý výskyt binárních čísel obsahujících dvě jedničky na sousedních pozicích, díky čemuž nyní dosahuje o přibližně 6-8 procent horších výsledků vůči 32 bitovému kódu v porovnání s Generalizovaným fibonacciho kódováním s $m = 3; m = 4; m = 5$. Účinnost Generalizovaným fibonacciho kódováním s $m = 3$ a $m = 4$ se na testovacím souboru pro tento rozsah ukázala srovnatelná.

4.1.4 Uniformní rozložení v rozsahu 65536 až 16777215

V tomto rozsahu už se značnou měrou projevují nedostatky zvolených kódovaní s proměnnou délkou kódu a zadaných parametrů. Nejlepších výsledků bylo dosaženo v případě Golombova kódu s $m = 10000000$, kdy došlo k úspoře 23,13 procent místa v porovnání s 32 bitovou reprezentací.

U Generalizovaného Fibonacciho kódu se zlepšení vůči 32 bitové reprezentaci pohybuje mezi 10,343 a 12,225 v závislosti na parametru m.

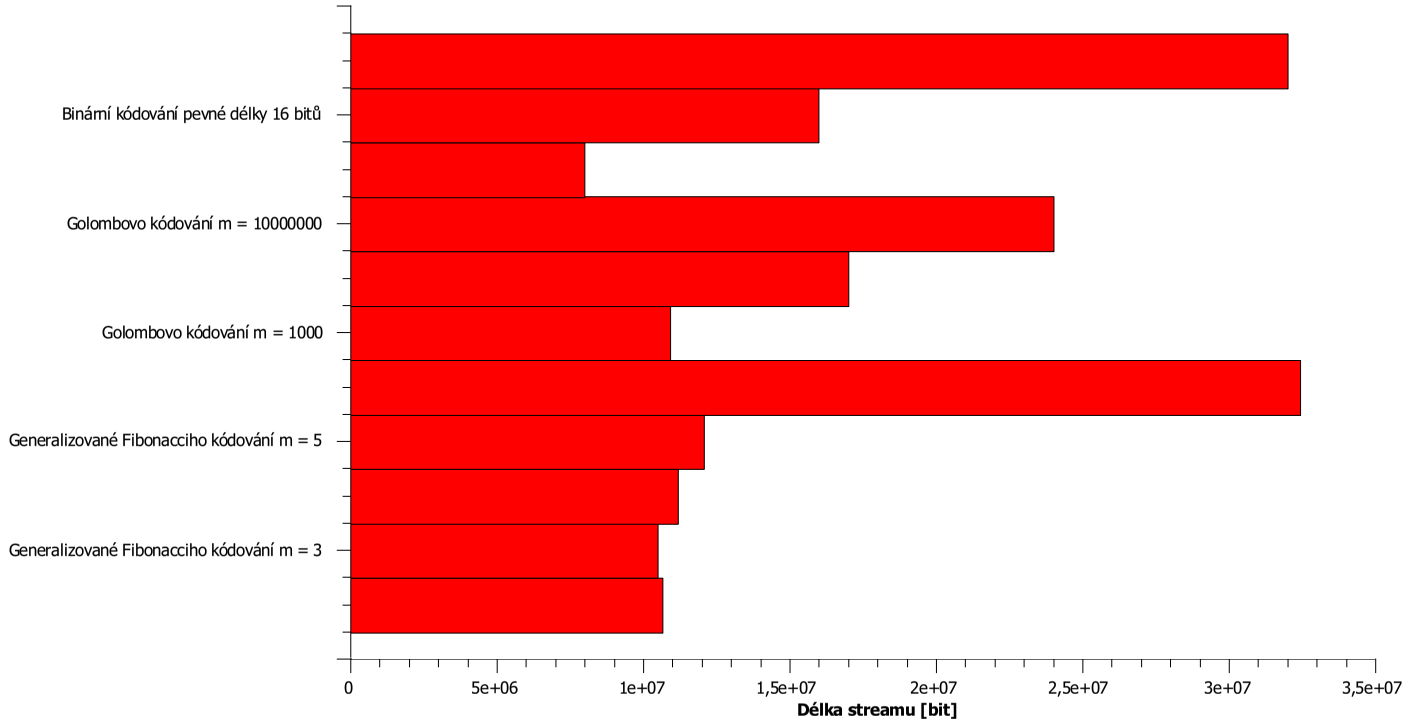
V případě Fibonacciho kódování a Golombova kódu s parametry $m = 1000$ a $m = 100000$ už nedochází ke kompresi, přičemž na Golombově kódování s $m = 1000$ vidíme, jaké následky může mít nevhodná volba velikosti parametru.

4.1.5 Uniformní rozložení v rozsahu 16777216 až 2147483647

V nejvyšším rozsahu už žádné z testovaných kódování neumožňovalo v porovnání s 32 bitovou reprezentací provádět komprimaci. Na Golombově kódu pak vidíme, jak závažné důsledky může mít nevhodně zvolený parametr, kdy u kodování s parametrem $m = 1000$ došlo k nárůstu délky o 3354458,503 procent.

Kódování	Délka streamu $[bit]$	Procentuální rozdíl vůči 32 bitovému číslu
Fibonacciho kódování	10 662 629	-66,679
Generalizované Fibonacciho kódování m = 3	10 498 121	-67,194
Generalizované Fibonacciho kódování m = 4	11 188 536	-65,036
Generalizované Fibonacciho kódování m = 5	12 062 750	-52,304
Goldbachovo kódování G0	32 436 377	1,364
Golombovo kódování m = 1000	10 909 799	-65,907
Golombovo kódování m = 100000	17 000 000	-46,875
Golombovo kódování m = 10000000	24 000 000	-25,000
Binární kódování pevné délky 8 bitů	8 000 000	-75,000
Binární kódování pevné délky 16 bitů	16 000 000	-50,000
Binární kódování pevné délky 32 bitů	32 000 000	0,000

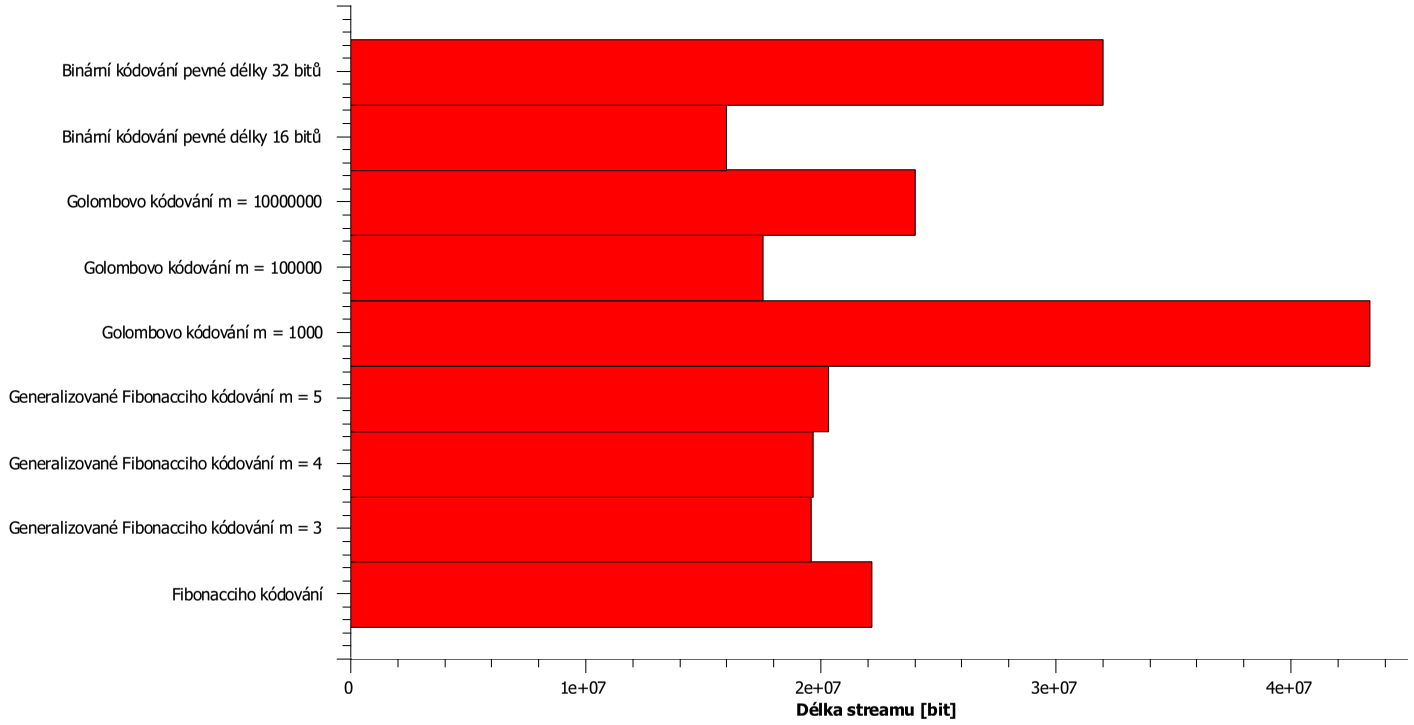
Tabulka 3: Délka streamů pro uniformní rozložení v rozsahu 1 až 255



Obrázek 2: Graf - Délka streamu pro uniformní rozložení v rozsahu 1 až 255

Kódování	Délka streamu $[bit]$	Procentuální rozdíl vůči 32 bitovému číslu
Fibonacciho kódování	22 192 175	-30,649
Generalizované Fibonacciho kódování m = 3	19 604 949	-38,735
Generalizované Fibonacciho kódování m = 4	19 677 211	-38,509
Generalizované Fibonacciho kódování m = 5	20 339 397	-36,439
Golombovo kódování m = 1000	43 370 264	35,532
Golombovo kódování m = 100000	17 527 711	-45,226
Golombovo kódování m = 10000000	24 000 000	-25,000
Binární kódování pevné délky 16 bitů	16 000 000	-50,000
Binární kódování pevné délky 32 bitů	32 000 000	0,000

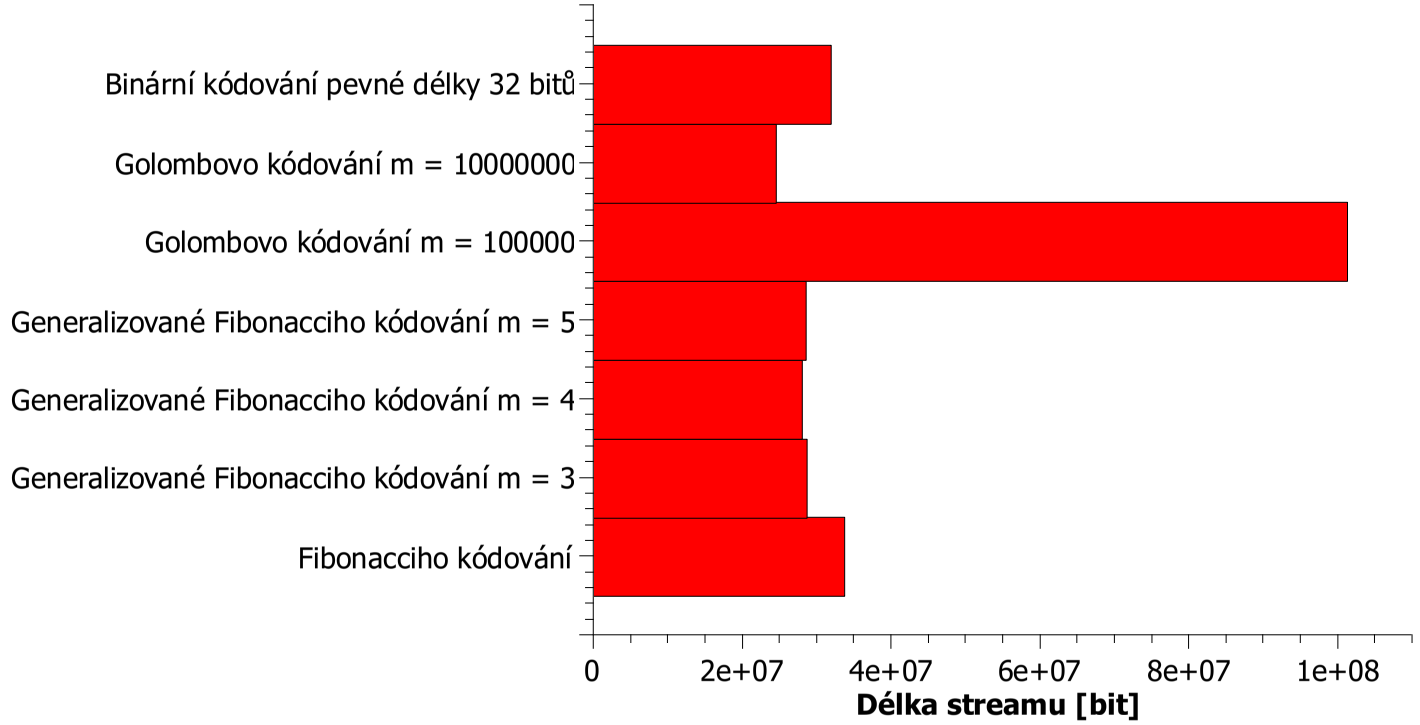
Tabulka 4: Délka streamů pro uniformní rozložení v rozsahu 256 až 65535



Obrázek 3: Graf - Délka streamu pro uniformní rozložení v rozsahu 256 až 65535

Kódování	Délka streamu $[bit]$	Procentuální rozdíl vůči 32 bitovému číslu
Fibonacciho kódování	33715518	5,360
Generalizované Fibonacciho kódování m = 3	28 690 139	-10,343
Generalizované Fibonacciho kódování m = 4	28 087 855	-12,225
Generalizované Fibonacciho kódování m = 5	28 556 700	-10,760
Golombovo kódování m = 1000	8 429 774 950	26243,047
Golombovo kódování m = 100000	101 382 364	216,820
Golombovo kódování m = 10000000	24 598 204	-23,130
Binární kódování pevné délky 32 bitů	32 000 000	0,000

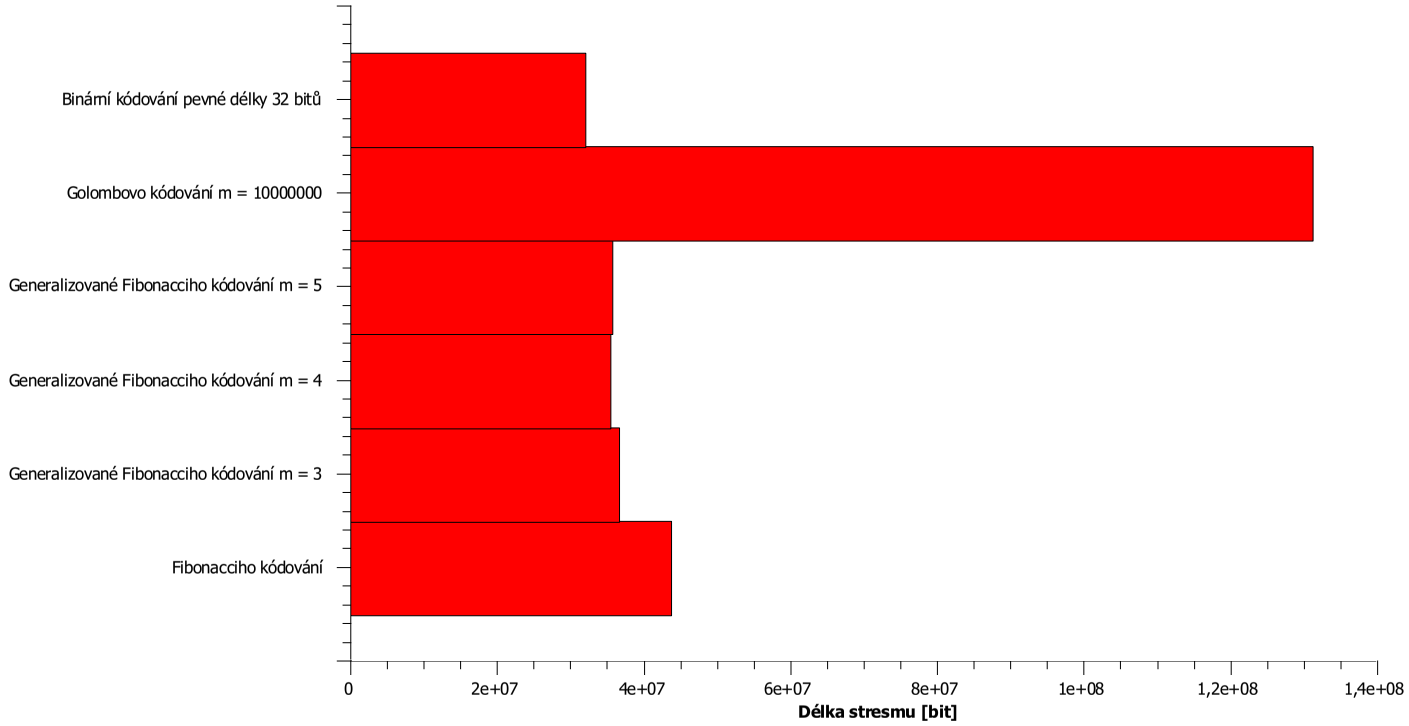
Tabulka 5: Délka streamů pro uniformní rozložení v rozsahu 65536 až 16777215



Obrázek 4: Graf - Délka streamu pro uniformní rozložení v rozsahu 65536 až 16777215

Kódování	Délka streamu $[bit]$	Procentuální rozdíl vůči 32 bitovému číslu
Fibonacciho kódování	43786535	36,832
Generalizované Fibonacciho kódování m = 3	36643596	14,511
Generalizované Fibonacciho kódování m = 4	35516350	10,989
Generalizované Fibonacciho kódování m = 5	35709096	11,590
Golombovo kódování m = 1000	1073458721101	3354458,503
Golombovo kódování m = 100000	10751672072	33498,975
Golombovo kódování m = 10000000	131167413	309,898
Binární kódování pevné délky 32 bitů	32 000 000	0,000

Tabulka 6: Délka streamů pro uniformní rozložení v rozsahu 16777216 až 2147483647



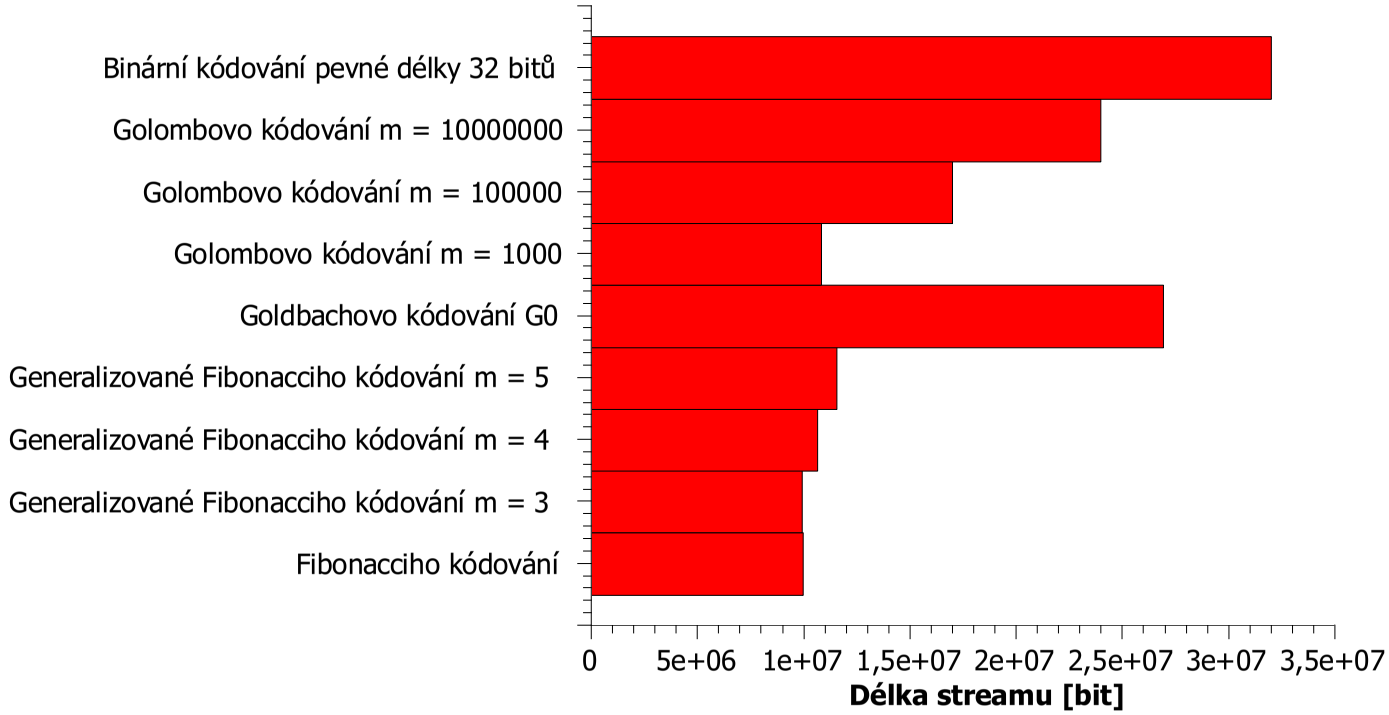
Obrázek 5: Graf - Délka streamu pro uniformní rozložení v rozsahu 16777216 až 2147483647

4.2 Normální rozložení

Při testování na souboru data_norm.txt prokázala všechny implementovaná kódování schopnost komprese. Jak vidíme v tabulce 7 nejhorších výsledků bylo dosaženo s Goldbachovým G0 kódováním, kdy bylo uspořeno 15,909 procenta místa. Nejlepších výsledků bylo dosaženo Generalizovaným Fibonacciho kódováním s parametrem $m = 3$. Velmi dobrých výsledků bylo dosaženo taktéž s Fibonacciho kódováním, Golombovým kódováním s parametrem $m = 1000$ a ostatními Generalizovanými Fibonacciho kódováními.

Kódování	Délka streamu $[bit]$	Procentuální rozdíl vůči 32 bitovému číslu
Fibonacciho kódování	9 955 665	-68,889
Generalizované Fibonacciho kódování m = 3	9 907 112	-69,040
Generalizované Fibonacciho kódování m = 4	10 654 092	-66,706
Generalizované Fibonacciho kódování m = 5	11 575 867	-63,825
Goldbachovo kódování G0	26 908 985	-15,909
Golombovo kódování m = 1000	10 851 087	-66,090
Golombovo kódování m = 100000	17 000 000	-46,875
Golombovo kódování m = 10000000	24 000 000	-25,000
Binární kódování pevné délky 32 bitů	32 000 000	0,000

Tabulka 7: Délka streamů pro normální rozložení



Obrázek 6: Graf - Délka streamu pro normální rozložení

5 Závěr

V rámci této práce jsem se seznámil s různými přístupy k binární reprezentaci celých čísel v rozsahu $< 1, 2147483647 >$ a implementoval třídy potřebné pro jejich nasazení v rámci kompresního frameworku.

Pokud to mé znalosti a schopnosti umožňovaly, snažil jsem se na místo "hrubé síly" využít pokročilejších přístupů. Jsem si ovšem vědom rezerv, které v tomto směru mé kódy stále ještě mají.

U Fibonacciho kódování se potvrdila vysoká efektivita pro malá čísla daná jednobitovým sufixem, kdy u testovacího souboru data_00_05.txt došlo k téměř 80 procentní úspoře místa v porovnání s 32 bitovým kódováním. S rostoucím rozsahem vstupních hodnot se ovšem efektivita vzhledem k častému výskytu binárních čísel s jedničkami na sousedních pozicích zhoršuje. Na testovacím souboru data_16_24.txt už nedocházelo ke komprimaci, ale naopak k nárůstu délky.

Generalizované Fibonacciho kódy pro $m = 3$, $m = 4$ a $m = 5$ vykazují v oblasti malých čísel poněkud horší výsledky v porovnání s Fibonacciho kódováním. S rostoucím rozsahem vstupních hodnot se ovšem stále více projevuje "hustší" binární reprezentace, která kompenzuje delší sufix, díky čemuž docházelo ke komprimaci ještě u testovacího souboru data_16_24.txt.

V případě Golombových a Riceových kódů se ukázala značná závislost účinnosti tohoto kódování na volbě parametru m. Nasazení těchto kódování se jeví jako nejvhodnější v případě "úzkého" intervalu vstupních hodnot, kdy zvolíme $m = x + 1$, kde x je horní uzávěrou intervalu.

Goldbachovo G0 kódování se ukázalo v rozsahu vstupních hodnot $< 1, 31 >$ jako rovnocenné ostatním testovaným kódováním. Pro jakýkoli větší rozsah už se bohužel ukázalo jako nepoužitelné kvůli průdkému nárůstu délky kódu, kdy k uchování čísla v řádu miliónů je potřeba více jak 100000 bitů, což by stačilo na uložení více jak 3000 čísel v daném rozsahu pomocí 32 bitové reprezentace.

Během testování se ukázala převaha 32 bitového kódu s pevnou délkou v rozsahu $< 16777216, 2147483647 >$, kdy zadaná kódování nedokázala dosáhnout lepších výsledků.

6 Reference

[1] Walder, Jiří, Krátký, Michael, Bača, Radim, Snášel, Václav, Platoš, Jan *Fast Decoding Alalgorithmus for Variable-Lengths Codes* [s.l.] , [2010].

[2] Salomon, David *Variable-length codes for Data Compression* Londýn, 2007.