# ВТУП

Обмін інформацією є однією з центральних ланок в роботі інформаційних систем. До інформації в таких системах ставляться наступні вимоги: об’єктивність, достовірність, повнота, точність, актуальність, корисність.

При передачі інформації через канали з’єднання або при збереженні на фізичні носії є ймовірність її пошкодження, що впливає на достовірність інформації. Через це інформація потребує захисту від помилок та фізичних вад каналів передачі або носіїв. Є багато різноманітних способів знаходження та виправлення помилок. При передачі інформації через мережу інтернет за допомогою протоколу TCP/IP вона поділяється на пакети, де кожен пакет на прикінці має хеш суму інформаційної частки пакету. В разі коли обчислена хеш сума пакету не співпадає з бажаною, іде повторна пересилка повідомлення. Тим самим користувач уникає помилок в отриманої інформації.

Але що робити коли повторна пересилка пакета неможлива або є надто повільною. В таких випадках доцільно використовувати коди виправлення помилок. Серед них доцільно виділити коди Хеммінга та коди Ріда-Соломона. Код Хемінга використовується для виправлення одиночної помлки. Код Ріда-Соломона дозволяє коректувати декілька помилок без повторної пересилки даних, та застосовується, наприклад, при збереженні даних на CD накопичувачі.

Ці коди є добре задокументованими та широко використовуються в електроніці, як на рівні ПЗ так і на рівні інтегральних схем. Але вони не підходять для виправлення великої кількості помилок. Наразі код Ріда-Соломона 15-11 дозволяє виправити 40 похибок на 10 блоків коду (600 біт), але, якщо вони локалізовані у різних блоках. Але вони перестають працювати при концентрації помилок в одному місці. Електричні реалізації цих кодів пришвидшують процес кодування/декодування, але їх не можно змінити, якщо параметри канала даних змінились.

Метою дипломної роботи є реалізація поміхо стійкого кодеку для збереження даних на фізичних носіях та в системах передачі даних де зустрічаються поміхи.

Для реалізації цієї задачі доцільно використовувати циклічні коди. Коди Хеммінга та коди Ріда-Соломона мають під собою однаковий математичний апарат – коди БЧХ, та обидва види кодів є частною реалізацією кодів БЧХ. Саме на кодах БЧХ доцільно реалізувати такий кодек. Це дозволить гнучко підстроювати коди під конкретну задачу.

Такий кодек біло би зручно використати для збереження інформації на фізичних носіях або для передачі інформації в космічний та військових галузях.

# АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

## 1.1 Введення в алгебру полів Галуа

На початку історії цифрової електроніки машини оперували машинними словами, котрі потім почали називати байтами. Байти в свою чергу реалізовувались як група бітів. Біт – мінімальна одиниця інформації, що може приймати два значення: 1 або 0. Були окремі спроби реалізувати трити – одинця інформації що може мати 3 значення: 0, 1 або2. Але таки системи не знайшли популярності, через свою повільність та складність в реалізації. З часом інженери виявили, що схеми котрі працюють в байтами, котрі мають кількість рівну цілим ступеням 2-ки працюють набагато швидше, ніж системи, з іншою кількістю бітів. Після років експериментів майже всі обчислювальні машини використовують байт довжиною 8 бітів.

Математичні операції над бітами є реалізацією булевої алгебри. Розділ математики що розглядає властивості таких типів даних називається – дискретна математика. Якщо роздивлятись машинне слово довжиною 8 бітів то ми побачимо, що їх варіативність кінцева, а саме є лише 28=256 варіантів байту. Що означає що люба інформація котрою оперують обчислювальні машини є комбінаціями цих байтів. Пошкодження цієї інформації означає – зміну бітів в байті на протилежний.

Циклічні коди також працюють на кінцевих полях. Хоча в загальному виді кінцеві поля реалізовують математику для різних основ та ступенів, для реалізації кодеку доцільно використати лише поля з основою – 2 та цілими ступенями, починаючи з 2-х

Поля Галуа є різновидами кінцевих полів, на основі котрих реалізовані циклічні коди. З визначення терміну «поле» - ми знаємо, що для поля існують детерміновані операції додавання, віднімання, множення і ділення. Також є операції відведення в ступінь, котру можна розглядати як окремий випадок множення.

Операції віднімання та додавання в полях Галуа є еквівалентними та представляють з себе додавання за модулем 2: (a+b)mod(2). Машинний еквівалент цієї операції є оператор XOR. Для цієї операції характерні наступні властивості: рефлективність, симетричність та транзитивність.

Операції множення має наступну реалізацію: знайти ступеневу форму чисел Галуа для множників; скласти ступені; знайти число відповідно до цього ступеня. Якщо ступень виходить за межі простору поля Галуа, то його слід поділити за модулем на максимальну ступінь цього поля.

Операція ділення робіться так аналогічно, але перед тим, ступінь дільника слід змінити на протилежну.

Маючи реалізацію цих 4-х операцій мі можемо кодувати бінарну інформацію в коди БЧХ.

Коди БЧХ (коди Боуза — Чоудхурі — Хоквінгема, Bose–Chaudhuri–Hocquenghem codes, BCH codes) – це клас циклічних кодів котрі використовуються для поміхо стійкого кодування інформації. Ключовим для цих кодів є те, що в них є інформація для виправлення заданої кількості помилок.

На відміну від коду Хеммінга або кодів Ріда-Соломона, БЧХ код можна налаштовувати на завдану довжину та кількість помилок(але с завданими обмеженнями).

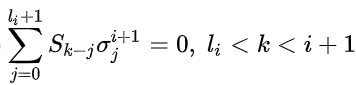
Для декодування кодів БЧХ є декілька алгоритмів, а саме:

1. Алгоритм Берлекемпа - Месі
2. Євклідів алгоритм
3. Пряме рішення (алгоритм Пітерсона - Горенстейна - Цирлера, ПГЦ)
4. Пошук Ченя
5. Алгоритм Форні

Роздивмося декілька алгоритмів.

Алгоритм ПГЦ: історично є першим алгоритмом декодування БЧХ коду. Цей алгоритм заснован на прямому рішенні системи поліноміальних рівнянь, де ведеться пошук коефіцієнтів локаторів помилок. Реалізація цього алгоритму підходить тільки для систем з маленькими полями Галуа.

Євклідів алгоритм. Через високу регулярніть популярний для рішення апаратного декодування кодів Ріда-Соломона.

Алгоритм Берклемпа-Месі. Є високопотожним алгоритмом. Його слід роздивлятись як ітеративний процес генерації реєстру ссуву, для генерації заданої послідовності синдромів. Його мета знайти найменшу ступінь поліному  котра буде задовільнять рівнянню . Його доцільніше використати для реалізації програмного декодеру.

## 1.2 Алгоритм створення БЧХ коду

* Визначити ступінь до поля Галуа, котре буде використовуватись для кодування.
* Построїти таблицю чисел Галуа, для завданої форми (десятина форма, двійкова форма, мультиплікативна форма.
* До цих чисел знайти ступеневу форму та мінімальний зворотній поліном.
* На основі попередніх обчислень, обчислити мінімальний зворотній поліном для коду БЧХ с завданою кількістю помилок. Знайти ступінь мінімального зворотного поліному.
* Знайти довжину інформаційного повідомлення, та поділити вхідну інформацію на блоки з цією довжиною.
* Закодувати інформацію блок за блоком, та зберегти її.

## 1.3 Алгоритм декодування БЧХ коду

* Визначити ступінь до поля Галуа, котре буде використовуватись для кодування.
* Построїти таблицю чисел Галуа, для завданої форми (десятина форма, двійкова форма, мультиплікативна форма.
* До цих чисел знайти ступеневу форму та мінімальний зворотній поліном.
* Взяти блок закодованої інформації.
* Обчислити синдроми. Якщо вони нульові – або інформація ціла, або помилки компенсували друг друга.
* Якщо синдроми не нульові – обчислити мінімальний полином локаторів помилок.
* Визначити ступінь цього пліному. Якщо вона перевищує кількість помилок, то декодування неможливе, якщо співпадає, або менша – продовжити декодування.
* Знайти корні поліному локаторів (методом повного перебору)
* Виправити вхідний поліном згідно до коренів. Перевірити новий поліном. Якщо синдроми дорівнюють нулю, то декодування пройшло вдало.

## Визначення вимог для кодеку

Так як кінцевим продуктом дипломної роботи має бути кодек для кодування/декодування кодів БЧХ, треба визначитись з тим, в якому виді цей кодек буде постачатись.

В класичному виді кодек має бути динамічною бібліотекою, котру можна підключити до програмного продукту, та використовувати з клієнтського додатку.

Також однією з вимог до кодеку є те, що він має бути легким для обчислювання на процесорі та не використовувати багато оперативної пам’яті.

1. **ОГЛЯД ІСНУЮЧИХ ПРОГРАМНИХ РІШЕНЬ ТА ТЕХНОЛОГІЙ**

## Порівняльні характеристики мов програмування

Насамперед треба означитись з мовою програмування. Згідно з вимог для кодеку та особливостей реалізації алегбри для кінцевих полів мова програмування має відповідати наступним характеристикам:

* Має бути достатньо швидкою.
* Має доволяти робити низькорівневу математику.
* Має мати весь перелік булєвих операцій.
* Використовувати мало оперативної пам’яті.

Згідно с характеристик можемо зробити висновок що мова програмування має бути достатньо низького рівня для ефективного використовування оперативної пам’яті та швидкого обчислення поліномів. З одного боку було б зручніше зробити програмну реалізацію на мові Python, так як математика там реалізована на векторах, що дозволяє нам працювати з дуже великими цифрами без переповнення, що характерні для низькорівневих статично типізованих мов програмування, але через векторну реалізацію математики, обчислення будуть проводитися довго. Також мова Python через особливості своєї реалізації є повільною, використовує багато оперативної пам’яті та працює лише в одному потоці, що наскладує обмеження на швидкість реалізації алгоритмів.

З статично типізованих мов програмування доцільно розглядати кандидатів серед C++, C# та Java, через те що вони є найпоширеніші серед мов для побудови клієнтських та серверних додатків.

Серед переваг Java та C# можна виділити наступні:

* Не має ручного менеджменту пам’яті.
* Багато бібліотек, котрі йдуть з SDK.
* Відносно легке створення коду, через попередні пункти.

Серед недоліків:

* Потрібних для роботи бібліотек не має, що нівелює користність існуючих бібліотек.
* Автоматичний менеджмент пам’яті приводить до великої кількості оперативної пам’яті в додатках.
* Є повільними відносно C++

Якщо розглядати мову C++, то можна побачити наступні переваги:

* Швидке виконання
* Доступ до низькорівневих операцій
* Можливість керувати користуванням оперативної пам’яті.
* Потужна реалізація стандартних колекцій/

Серед недоліків:

* Складна реалізація через специфічний набор стандартних бібліотек (але в нашому випадку, їх досить)
* Великий поріг входу для програмістів.

Зваживши всі плюси і мінусі мов програмування було вирішено використати мову C++.

## 2.2 Огляд існуючих програмних рішень

Через складну реалізацію кодів БЧХ та літературу до неї, котра орієнтована більше на науковців теоретиків гарних універсальних імплементацій БЧХ алгоритму для двійкового коду в відкритому доступі не має.

В реалізація полів Галуа в відкритому доступі більшість реалізована для GF(23) або GF(24), що не є гнучким. Також в цих реалізаціях не має мінімальної зворотної функції для GF.

## 2.3 Порівняльні характеристики способів доставки коду

Кодек для кодування декодування треба використовувати в клієнтських додатках як сторонню бібліотеку. Згідно з принципами побудови бібліотек в мові С++ є 3 способи це зробити:

* Створити дінамічну бібліотеку.
* Створити статичну бібліотеку.
* Віддати код всього пакету для компілювання код як частину додатку.

Розглянем переваги та недоліки кожного варіанту.

**Динамічна бібліотека.**

Переваги:

* Інкапсуляція коду бібліотеки, та захист її від модифікацій.

Недоліки:

* Клієнтська програма має бути скомпільована с тими же ключами що і динамічна бібліотека. Через це треба мати інфраструктуру для побудови бібліотеки під всі бажані платформи, та версії операційних систем.
* Складність в підключені динамічних бібліотек для новачків.

**Статична бібліотека.**

Переваги:

* Інкапсуляція коду бібліотека, та захист її від модифікацій.

Недоліки:

* Трохи легша в підключенні ніж динамічна бібліотека.

**Доставка коду пакетом.**

Переваги:

* Легкий спосіб доставки коду.
* Легкий спосіб підключення коду в проект.
* Легка реалізація кросплатформеності коду.
* Компіляція коду з бажаними ключами.
* Клієнт може подивитися в код та модифікувати його в разі потреби на це, що дозволяє розробляти менший об’єм документації

Недоліки:

* Клієнт може модифікувати та передивлятись код, що може бути критичним для бізнесу.

Дивлячись на ті обмеження котрі стоять перед нами в рамках дипломної роботи, біло вирішено доставляти код, як набір файлів, котрі треба самостійно включати до проекту.

2.4 Огляд існуючих алгоритмів декодування

Як вказувалось раніше є декілька методів декодування БЧХ коду. Розглянемо ближче кожен з них.

**Евклідів алгоритм.**

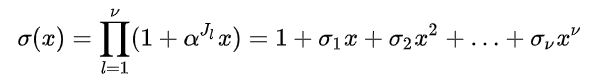
Евклідів алгоритм – в нашому випадку це класичний евклідів алгоритм, проте замісць знаходження найбільшого спільного дільника (НСД), відбувається пошук НСД для двох поліномів.

Прийняте кодове слово *r(x)=c(x)+e(x)*, де *e(x)* – поліном помилок, а *c(x)* кодове слово + породжуючий поліном для перевірки. Згідно з цим в нас може бути *u≤t=(d-1)/2* помилок на позіціях *i1,i2,…, iu*, де *t* максимальна кількість помилок, котрі можна виправити. З цього слідує *e(x)=ei1xi1+ ei2xi2 +ei3xi3+…+ eiuxiu,* а *ei1, ei2, …, eiu –* це помилки передачі.

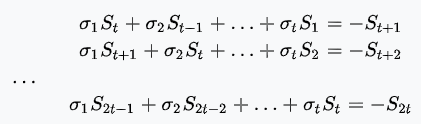
Визначимо поліном значення помилок *Λ=σ(x)S(x).* Де синдромний поліном дорівнює:



Згідно з визначенням локаторів помилок, він дорівнює:



де корені рівняння є зворотними величинами локаторів помилок. Тоді буд вірне наступні відносини між коефіцінтами полінома локаторів помилок і синдромами:



Це рівняння є основним та лежить в основі декодування БЧХ коду. Згідно з основної системи рівняннь:



Задача зводиться к тому щоби знайти всі *Λ(x)* так, що би корені не були вищи за td. По суті це є розширений алгоритм Евкліда. Має комплексну складність O(n3).

**Алгоритм ПГЦ**.

Алгоритм ПГЦ - алгоритм заснован на прямому рішенні системи поліноміальних рівнянь, де ведеться пошук коефіцієнтів локаторів помилок. Це алгоритм прямого рішення системи рівнянь, де ми з одного боку маємо обчислені синдроми, а з іншого боку нам відома максимальна кількість помилок котрі мі можемо виправити.

Нехай БЧХ код на полем GF(q) довжини n і конструктиною відстаннь d задається породжуючим поліномом g(x), який серед своїх коренів має елементи:



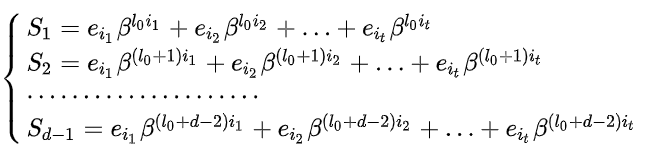
де l0 ціле число (наприклад 0 або 1).

Щыгдно з основним рывнянням можна виділити *j-*й синдром Sj прийнятого повідомлення *r(x):*

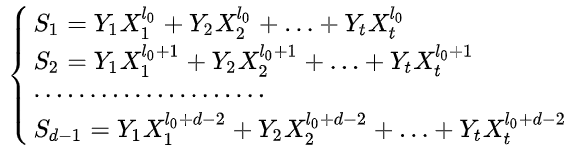
**

Задача полягає в тому, що треба знайти всі комбінації кількості помилок, та їх позіцій (через те що, в контексті дипломної роботи мі розглядаємо лише бінарні коди, то значення помилки завжди 1).

Припустимо, для початку, що *u* точно дорівнює *t*. Запишемо систему лінійних рівнянь:



Визначим *Xk=bik* локатор *k-ї* помилки, а через *Yk=eik* зачення помилки, *k=1,2, ..., t.*

**

Зробимо поліном локаторів помилок:



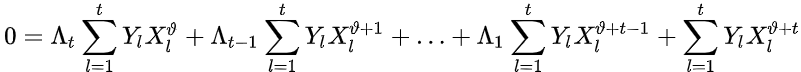
Коренями цього поліному елементи, зворотні локаторам помилок. Помножимо обидві сторони на . Отримане рівняння буде справедливо для *ν=l0, l0+1, l0+2, l0+2, …, l0+d-1, l=1, …,t:*

**

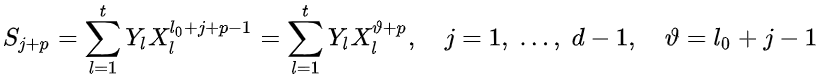
Помножимо , таким чином отримаємо:



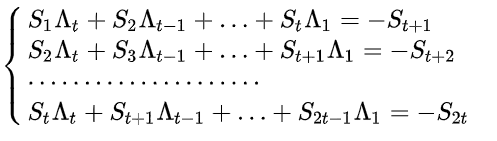
Приведемо рівняння до виду:



Враховуючи що:



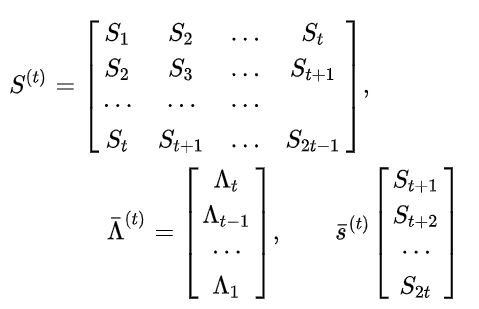
Таким чином, можемо отмиати систему лінійних рівнянь:



Або в матричній формі:



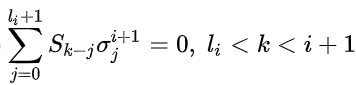
Де:



Якщо кількість помилок дорівнює *t,* то система вирішається для коєфіцієнтів *Λ1, Λ2, Λ3, …, Λt*. Якщо ні визначник матриці *S(t)* системи буде дорівнювати 0. Що значить, що кількість помилок меньша. Тому слід повторити процес з кількістю помилок *t-1,* і так далі до 1.

Як ми бачимо такий алгоритм є складним для побудови в програмному виді, та дуже затратним за ресурсами.

Євклідів алгоритм. Через високу регулярніть популярний для рішення апаратного декодування кодів Ріда-Соломона.

Алгоритм Берклемпа-Месі. Є високопотожним алгоритмом. Його слід роздивлятись як ітеративний процес генерації реєстру ссуву, для генерації заданої послідовності синдромів. Його мета знайти найменшу ступінь поліному  котра буде задовільнять рівнянню .

**Алгоритм Берлекемпа – Мессі.**

Алгоритм Берлекемпа – Мессі – його ми будемо реалізовувати, тому розглянемо його далі. Слід зазначити, що цей алгоритм найшвидший серед інших алгоритмів декодування, тому саме його доцільно вживати для декодування.

# ПРОЕКТУВАННЯ ДОДАТКУ

Додаток доцільно поділити на 3 окреми частини: модуль арифметики полів Галуа, кодер БЧХ, та декодер БЧХ.

Перша частина буде реалізовувати поліноміальну алгебру для заданих кодових слів розміром q ( GF(q)). В цій частині треба реалізувати таблицю алфавіту векторну форму, алгоритмічно адаптивну форму та мультиплікативну форму GF(q), та знайти зворотній поліном.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **Векторне подання** | **Адитивна група** | **Ступенева форма** | **Зворотній поліном** |
| 0 | 0 | 0 | - |  |
| 1 | 1 | 1 | x0 | x+1 |
| 2 | 10 | x | x1 | x4+x+1 |
| 3 | 11 | x+1 | x4 | x4+x+1 |
| 4 | 100 | x2 | x2 | x4+x+1 |
| 5 | 101 | x2+1 | x8 | x4+x+1 |
| 6 | 110 | x2+x | x5 | x3+x+1 |
| 7 | 111 | x2+x+1 | x10 | x3+x+1 |
| 8 | 1000 | x3 | x3 | x4+x3+x2+x+1 |
| 9 | 1001 | x3+1 | x14 | x4+x3+1 |
| 10 | 1010 | x3+x | x9 | x4+x3+x2+x+1 |
| 11 | 1011 | x3+x+1 | x7 | x4+x3+1 |
| 12 | 1100 | x3+x2 | x6 | x4+x3+x2+x+1 |
| 13 | 1101 | x3+x2+1 | x13 | x4+x3+1 |
| 14 | 1110 | x3+x2+x | x11 | x4+x3+1 |
| 15 | 1111 | x3+x2+x+1 | x12 | x4+x3+x2+x+1 |
| 1 | 1 | 1 | x15 |  |

Для побудви БЧХ коду треба реалізувати повну арифметику для чисел полів Галуа, а саме 4 арифметичні дії: додавання, віднімання, множення та ділення, так як ці арифметичні дії будуть використовуватись, при кодуванні та декодуванні БЧХ. Також для реалізації кодування та декодування, треба зробити арифметику множення чисел Галуа на невизначені коефіцієнти.

Наступний модуль програми, кодер БЧХ: користувач має зазначити розмірність кодового повідомлення, та кількість помилок, що слід виправити в кодері. Також на вході в цей модуль має буде повідомлення що слід закодувати, або декодувати. Декодер має бути реалізацією алгоритму Берклемпа-Месі.

Для тестової програми, достатньо вносити помилки власноруч, щоби перевірити, здатність декодеру локалізувати ці помилки, та правильно їх обробляти.

## Реалізація алгебри для чисел полів Галуа

Скінченне поле або поле Галуа – це поле, яке складається зі скінченної множини елементів.

Найменше поле Галуа *GF(2)=F2* складається тільки два елементи, 0 та 1, арифметичні операції на котрих проходять як к класичній алгебрі, за винятком 1+1=0.

Поля Галуа задаються як *GF(q)*, де *q* деяке число, що є ступенем деякого простого числа за деяким ступенем *m: cm=q.* В такому полі елементи будуть починатись з 0 и до *q-1.*

Ідея застосування полів Галуа за основою 2 (*GF(2m)*) складається в тому, що доцільно розглядати елементи такого поля як елементи алгебраїчної структури – векторного простору над цим полем.

Перед тим як розглядати ознаки поля, розглянемо елементи абелевої групи.

**Абелева група** або **комутативна група** – це група, в котрій математичні операції задовольняють умова комутативності. Наприклад, *x\*y=y\*x,* або *x+y=y+x*. Якщо це ствердження не виконуються, то таки операції називаються некомутативними. Група операцій які не є комутативними складаються з таких операцій, як виднимання (a-b), ділення(*a/b*), піднесення до степеня (ab), композіція функцій (*f(g(x)*).

**Кільце** – яка є абстрактною множиною, і є розширенням над абелевою групою. Кільцем називають таку множину в якої є дві операції: додавання (+), та множення (сусдньо расположення твух елементів). Для всіх кілець ці операції мають наступні характеристики:

* відносно додавання (+) R є абелевою группою;
* замкнутіть: множення *ab* ϵ *R* для любих *a* та *b* що лежать в *R;*
* закон ассоціативності: *a(bc)=(ab)c*;
* закон дистрибутивності: *a(b+c)=ab+ac, (b+c)a=ba+ca.*

**Поля ­–** це розширення над кільцем до додавання, віднімання, множення ще додається операція добуток.

Поле має наступні властивості:

* множина складає абелеву групу по додаванню та відніманню;
* це поле замкнуто відносно множення, а группа елементів без властивості нуля складає абелеву групу;
* закон дистрибутивності.

Властивості нуля: *a\*0=0*, *a+0=a*. На нуль ділити заборонено.

Властивості одиниці: a\*1=a.

В контексті дипломної роботи будуть роздивлятись тільки скінченні поля – тобто поля кількість елементів котрих скінченне.

Розглянемо математичні операції над елементами поля Галуа.

**Операція плюс**.

Ця операція є еквівалентом операції плюс за модулем 2: a ⊕b = c.

Наприклад:

10 + 7 = x3 + x + x2 + x + 1 = x3 + x2 + (1 + 1)x+1= = x3+ x2 + 1 = 13.

(10 + 7)16 = (1010)2 + (0111)2 = (1101)2 = (13)16.

З даного прикладу мі бачимо, що ця функція є еквівалентом бінарної операції XOR.

**Операція мінус.**

Повністю співпадає з операцією плюс.

**Операція множення.**

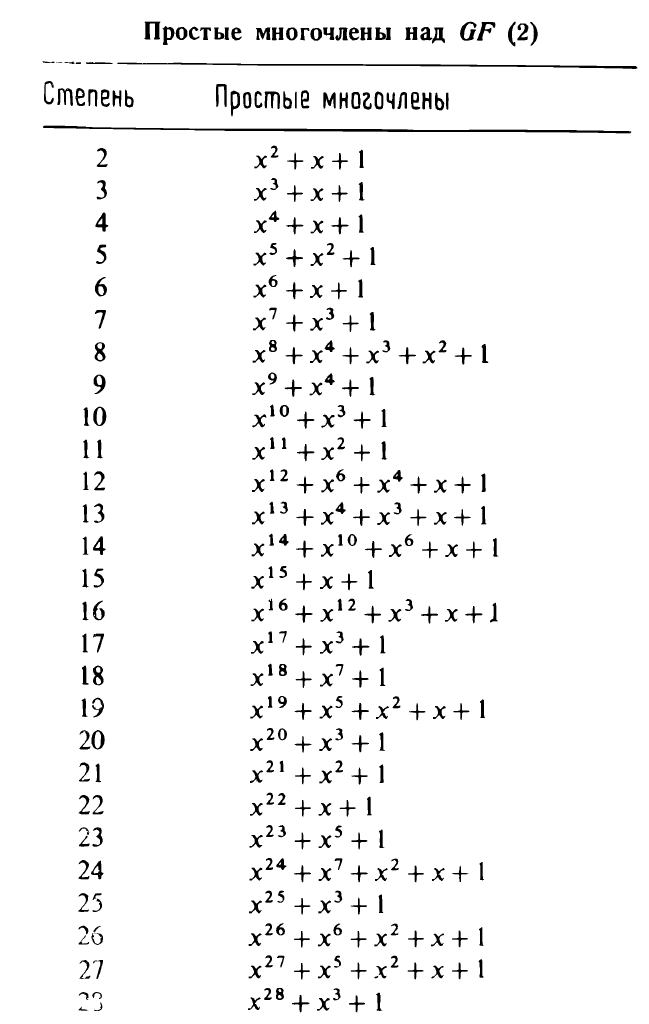
Для елементів поля Галуа, операція множення відбувається за наступним алгоритмом: *ai\*aj=a(i+j)mod(q)*. Як мі бачимо за алгоритму множення, то операція множення: це операція складання ступенів числа за модулем *q.*

**Операція добутку.**

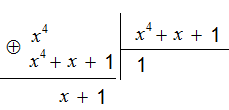
Операція добуту схожа на операцію множення, але на першому етапі ми множимо ступень дільника на -1 та ділемо за модулем *q.* Тобто *ai/aj= ai/a(j\*-1)mod(q),* далі робимо множення.

Як ми бачили вище операції множення так операція добутку оперує ступенями елементу поля. Для того щоб знайти ступень елементу поля, треба взяти всі існуючи ступені поля та їх поділити на примітивний поліном ступеня. Залишок від такого ділення і буде елемент поля, котрому ця ступінь належить.

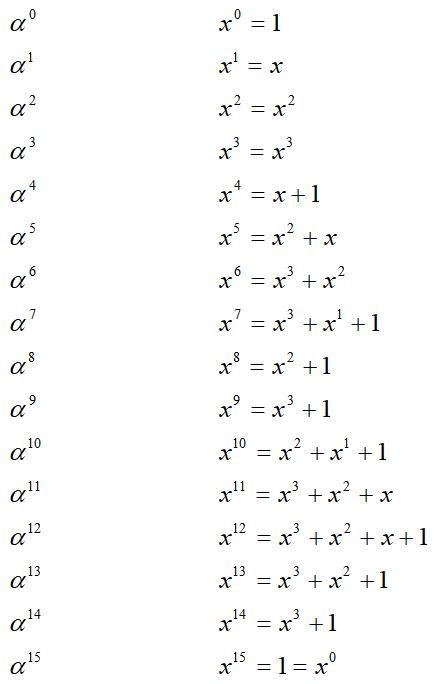
Ці прості поліноми завжди мають ступінь *q+1.* На рисунку нижче показані прості поліноми до різних ступенів.



Приклад проведення операції добутку для знаходження ступеню в GF(16):



Якщо провести цю операцію для *GF(16)* то ми побачимо наступну картину:



Після цієї операції множення і добуток зробити легко.

**Операція зведення ступень.**

Ця операція дуже просто, якщо нам вже відома ступень елементу поля. Для операції зведення в ступень треба перемножити ступені за модулем *q-1*: *(a i ) j = a (i· j)mod(q-1).*

## Реалізація алгоритму знаходження мінімальної зворотної функції

Для щоби закодувати повідомлення в БЧХ код, нам треба зробити реалізацію ще однієї операції. Треба знайти мінімальну зворотню функцію.

Мінімальна зворотна функція – це функція, помножив на котру в результаті ми отримуємо 0. Її ступінь завжди на 1 вища, ніж ступінь нашого поліному. Тобто для *GF(q)* ступінь мінімальної зворотної функції *q+1*.

Алгоритм знаходження мінімальної зворотної функції.

По-перше треб знайти циклотомічний клас для заданого ступеню.

Циклотомічним класом називається група в котрій елементи сполучени. Циклотомічний клас знаходиться по наступній формулі:



Почнемо з елементу GF(16) *а0*. Згідно до формули, його циклотоміний клас:



Так як всі елементи поля є однаковими, то його циклотомічний клас {0}.

Елементу GF(16) *а1*. Згідно до формули, його циклотоміний клас: {1, 2, 4, 8}. Підставимо елементи до формули.



Елементу GF(16) *а2*. Його циклотомічний клас та мінімальну зворотну формулу вже знайдено. Також елементи 4, 8 мають такий самий поліном.

Елемент GF(16) *а3.* Згідно з формулою С = {3, 6, 12, 24}. Так як 24 більше основи нашого поля 16-ти, то треба його поділити на *q-1*: *24 mod 15=9.*

Згідно з цим: m3(x) = (x – a3 ) (x – a6 ) (x – a9 ) (x – a12 )= x4 + x3+ x2 + x1+1.

Цей поліном вірний для 3, 6, 9, 12.

Елемент GF(16) *а5*. Його циклотомічний клас: С = {5, 10, 5, 10}. Так як є повторюючи елементи то скоротимо циклотомічний клас до : С = {5, 10}.

Його мінімальна зворотня функція m5(x) = (x – a5 ) (x – a10 ) = x2 + x+1.

Елемент GF(16) *a7*. Його циклотоміний клас С = {7, 14, 28, 56} = {7, 14, 11, 13}.

Мінімальний поліном: m7(x) = (x – a7 ) (x – a14 ) (x – a11 ) (x – a13 ) = x4 + x3+1.

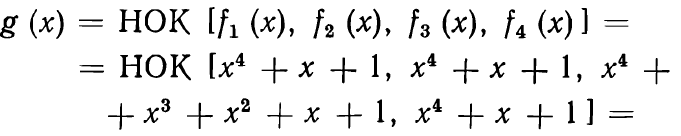
Як ми бачимо таким чином ми отримали мінімальні поліномі для всіх елементів GF(16).

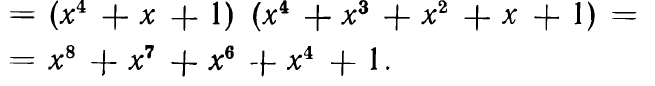
## Реалізація кодування БЧХ

БЧХ код – є циклічним кодом виправляючим пакети помилок. Пакети помилок доцільно розглядати як поліном *e(x)=xib(x)(mod xn-1),* де *b(x)* поліном ступеню не вище за *t-1*, де *t –*  максимальна кількість помилок що може виправити БЧХ код.

Виберемо розмір поля Галуа GF(16). Визначемо максимальну кількість помилок що можна виправити t=2.

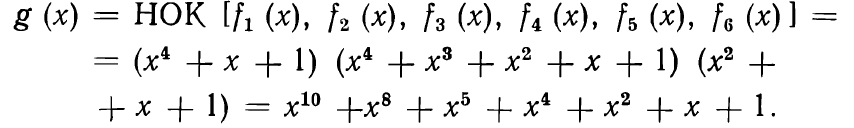
Після цього треба побудувати породжуючи поліном виправляючи 2 помилки. Для цього треба знайти НОД (найменший спільний дільник) зворотних поліномів для перши 2\*t елементів: *а1, a2, a3, a4.*





Оскільки ступень *g(x)* дорівнює 8, а довжина нашого коду *q=15*, q-k=8. Звідси k = 7 довжина інформаційної частки. Тобто мі отримали породжуючий поліном для коду БЧХ (15,7) (це один з кодів Ріда-Соломона).

Побудуємо породжуючий поліном для t=3 помилок.



Це породжуючий поліном для БЧХ(15,5). Можна збільшувати t, поки довжина породжуючого поліному буде менша за q-1.

Тепер можна повністю описати алгоритм кодування БЧХ коду:

1. визначити розмір поля;
2. визначити кількість помилок;
3. перевірити довжину породжуючого поліному, якщо вин менші за розмір бажаного поля Галуа, то можна йти далі, якщо ні, то зменшити кількість помилок, поки довжина породжуючого поліному не буде менша ніж розмір простору;
4. повідомлення з інформаційного каналу поділити на довжину кодового слова; залишок заповнити нулями;
5. зробити ссув блоку кодового слова, на погрожуючого поліному;
6. поділити отриману інформацію на породжуючий поліном;
7. залишок додати до кодового слова з ссувом.

Після всіх цих операцій, ми отримаємо закодоване повідомлення, з інформацією як відновити t – кількість помилок. Згідно з теорією вище, таке повідомлення ділиться на породжуючий поліном без залишку, якщо в повідомленні не були ніяких помилок, так як кодове слово з ссувом давало залишок від добутку на породжуючий поліном, то ми його компенсували згідно до алгебри циклічних кодів. Закодоване повідомлення треба повернути користувачу, котрий його буде відправляти адресат або зберігати його на диск, тим самим інформація попаде в середу в котрій можливо внесення помилок з часом.

* 1. Реалізація декодування БЧХ

Кодування кодового слова є математично простою операцією порівняно з декодуванням.

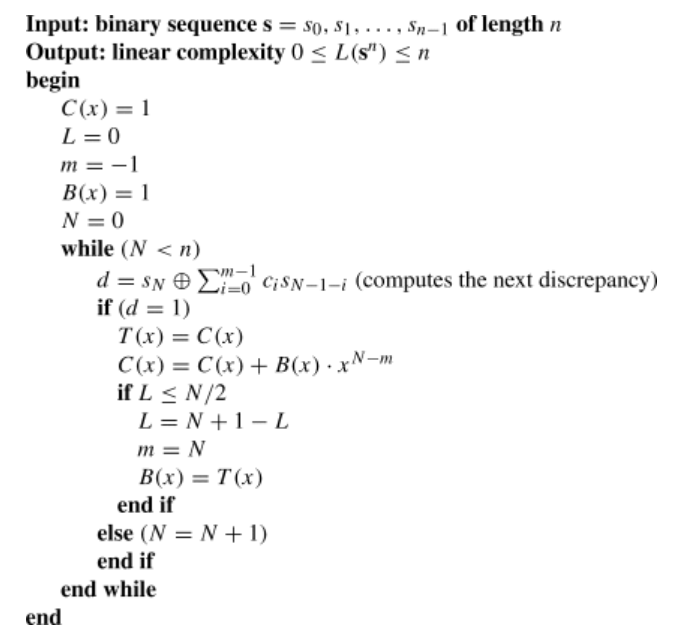
На першому етапі декодування, треба поділити вхідне повідомлення (кодове слово) на породжуючий його поліном. Якщо залишок від добутку дорівнює нулю, то з великою достовірністю можна казати, що повідомлення надійшло без внесення помилок. Про те, можливі випадки, що внесені помилки компенсують залишок від добутку, але така вірогідність є дуже маленькою.

Якщо результат добутку нульовий, то треба вичитати старші n бітів з кодового слова, та повернути їх користувачеві. Після цього можна обробляти наступне кодове слово.

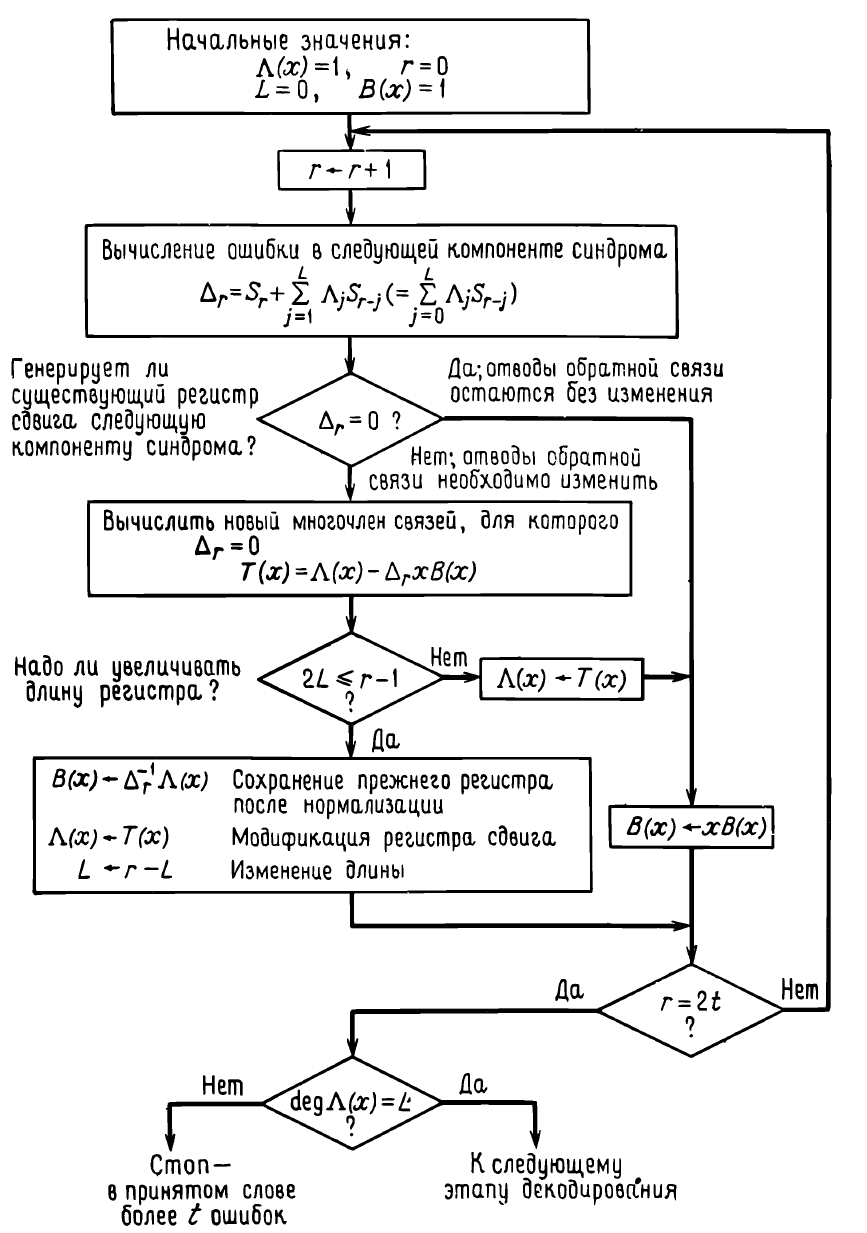
Якщо в результаті добутку є залишок, то треба перейти до процедури відновлення даних.

Для процедури відновлення даних в рамках дипломної роботи буде використано алгоритм Берлекемпа – Мессі (BMA). Це швидкий алгоритм, котрий дозволяє сконструювати поліном локаторів помилок. Алгоритм Берлекемпа – Мессі не знаходить самі локатори, тому після знаходження поліному локаторів помилок, треба ще знайти його корені, але це легко зробити прости перебором.

Реалізація алгоритму псевдокодом:



Блок схема реалізації алгоритму:



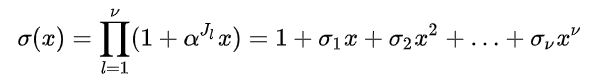
Роздивимося теорію, що стоїть за цим алгоритмом.

Прийняте кодове слово *r(x)=c(x)+e(x)*, де *e(x)* – поліном помилок, а *c(x)* кодове слово.

Визначимо поліном значення помилок *Λ=σ(x)S(x).* Де синдромний поліном дорівнює:



Згідно з визначенням локаторів помилок, він дорівнює:



Мета цього алгоритму, як і алгоритму це створення поліному локаторів помилок, котрий потім можна розв’язати алгоритмом Евкліда. На відміну від інших алгоритмів декодування БЧХ, алгоритм BMA зробити це з мінімальною кількості кроків та обчислювань.

Роздивимося як реалізувати цей алгоритм.

На першому кроці, треба вирахувати синдроми помилок, в кількості 2\*t. Для цього ми побудуємо синдромний поліном:

s(x) = 1 + s1x + s2x 2 + · · · + s2tx 2t

Заповнити два поліноми Λ(x), B(x) нулями, окрім першого елементу, котрий треба ініціалюзувати 1. Також треба зробити допоміжні змінні L = 0, r = 0.

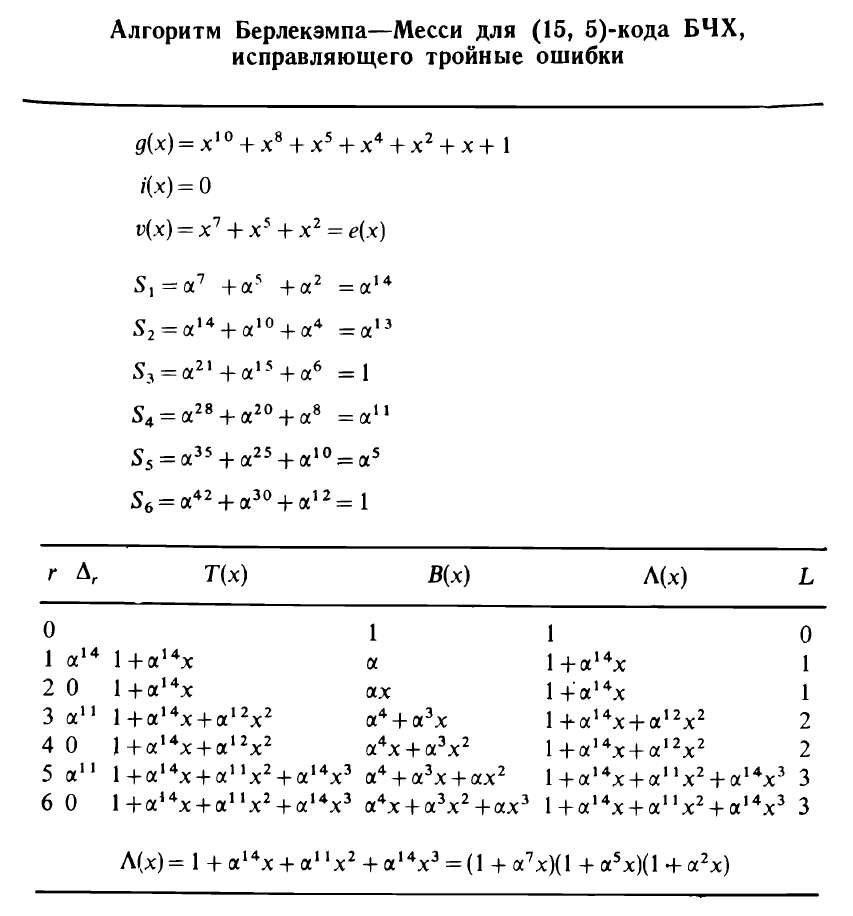
В цікли від r = 1 до r = 2\*t робимо обчислити поліном регістру ссуву:

Δr ←Sr +

В залежності від обставин змінюємо Λ(x) та B(x) на відповідні значення за блок схемою.

Якщо наприкінці циклу deg (Λ(x)) = L, то виправлення помилки можливе, якщо ні, то повідомлення декодувати цим способом неможливо.

Етапи декодування BMA виглядають наступним чином:



Після того, як отримано поліном локаторів помилок, треба знайти всі корені поліному локаторів помилок. Корені поліному локатору є зворотними до позицій помилок. Тим самим чином, робимо ссув на зворотною ступінь поліному, та операцію додавання за модулем 2.

Все, ми отримали кодове слово, з виправленими помилками. Можемо перевірити залишок від добутку на породжуючий поліном. Якщо залишок нульовий, то ми пройшли верифікацію кодового слова.

Після того як декодер готов можна перейти до стадії тестування.

# ТЕСТУВАННЯ

Для тестування кодеку, треба створити тестовий проект з підтримкую мови С++11, та додати файли кодеку до нього: "GaloisFieldNumber.h", "BCH\_coder.h", "MultXA.h", "GaloisFieldNumber.cpp", "BCH\_coder.cpp", "MultXA.cpp", “PrimitiveGroups.h”.

Після підключення пакету кодека можна розпочати тестування.

На першому треба створити всі ненульові елементи поля Галуа. Наприклад, наступний код для GF(16):

int n = 4;

std::vector<GaloisFieldNumber> numberVec;

std::vector<std::string> numberVectorVec;

int nSize = static\_cast<int>(pow(2, n));

for (int i =0; i < nSize; ++i)

{

if (GaloisFieldNumber::CheckGaloisParam(n, i))

{

numberVec.push\_back(GaloisFieldNumber(n, i));

}

else

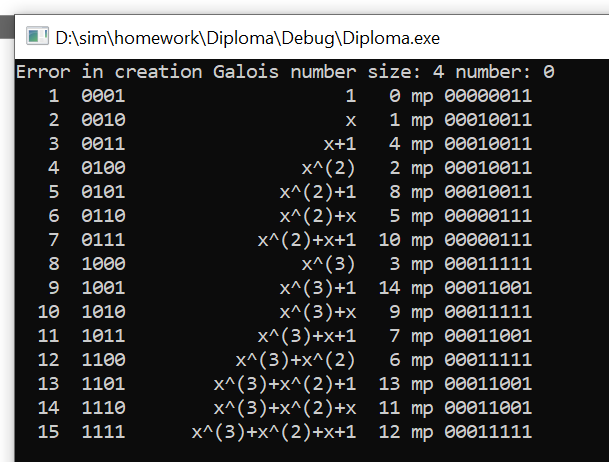
{

std::cerr << "Error in creation Galois number size: " << n << " number: " << i << std::endl;

}

}

Роздрукуємо повну таблицю результатів:



В перший колонці, десятковий вид числа, в другій двійковий, в третій алгебраїчно адитивний, в 5-й ступеневий, остання колонка відповідає за мінімальний зворотній поліном в двійковому виді. Порівняємо їх з еталонними таблицями. Всі значення співпадають.

Після цього треба перевірити математичні операції. Наступний код для перевірки операцій додавання, віднімання, множення та добутку:

GaloisFieldNumber gf = GaloisFieldNumber(4, 2) + GaloisFieldNumber(4,6);

std::cout << " GF(4) 2+6 = " << gf.getNumber() << std::endl;

gf = GaloisFieldNumber(4, 2) + GaloisFieldNumber(4, 6);

std::cout << " GF(4) 2-6 = " << gf.getNumber() << std::endl;

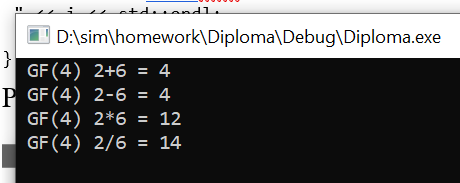
gf = GaloisFieldNumber(4, 2) \* GaloisFieldNumber(4, 6);

std::cout << " GF(4) 2\*6 = " << gf.getNumber() << std::endl;

gf = GaloisFieldNumber(4, 2) / GaloisFieldNumber(4, 6);

std::cout << " GF(4) 2/6 = " << gf.getNumber() << std::endl;

Результат роботи коду:



Перевіримо порахувавши власноруч. Значення співпадають.

Протестуємо створення числа Галуа зі ступені. Наступний код для перевірки цього:

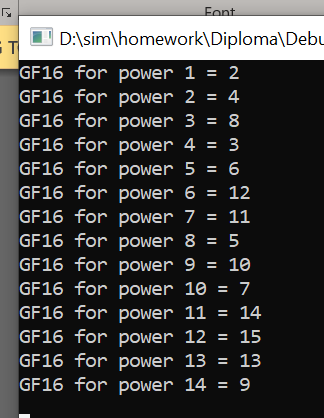
for (int i = 1; i < 15; ++i)

{

std::cout << "GF16 for power "<< i <<" = " << GaloisFieldNumber(4, GaloisFieldNumber::GetGaloisNumberFromPower(4, i)).getNumber() << std::endl;

}

Результат роботи коду:



Перевіримо порахувавши власноруч. Значення співпадають.

Після цього треба перевірити роботу кодеку без внесення помилок. Наступний код це реалізовує:

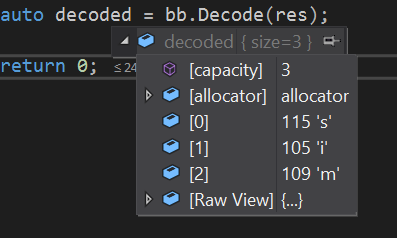
std::vector<unsigned char> sim = { 's','i','m'};

BCH\_Codec bb (4, 2); // GF(2pow4), error t = 2

auto res = bb.Encode(sim);

auto decoded = bb.Decode(res);

Результат роботи коду:

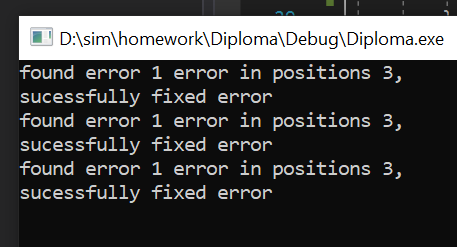


Як ми бачимо повідомлення декодовано успішно.

Тепер підправимо код для декодера, і введимо похибку в кожном 3-ому молодшому біті блоків коду:

tryToWithDecodeErrors(inMessage^(1<<3));

Результат роботи коду:

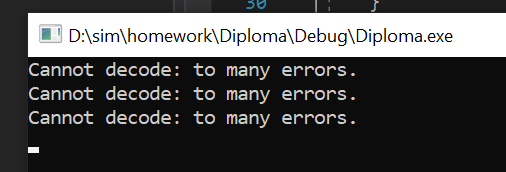


Якщо переглянути декодоване слово, то можна побачити що воно не змінілось.

Тепер підправимо код для декодера, і введимо похибку в кожном 3-ому молодшому біті блоків коду:

tryToWithDecodeErrors((((inMessage^(1<<3))^1<<10)^1<<15));

Результат роботи коду:



Якщо переглянути декодоване то видно що воно пусте.

# Інструкція для програміста

Кодек поставляється як набір файлі що треба додати до свого с++11 проекту. Для цього необхідно зберегти пакет на диску, та прописати в прописати “Include search path” до каталогу на дисці де збережен пакет, а також додати "GaloisFieldNumber.cpp", "BCH\_coder.cpp", "MultXA.cpp" у “Build targets”. Або якщо користуватися IDE додати файли пакету до свого проєкту.

Після цього в файлах що будуть використовувати декодер додати до хедерів:

#include "GaloisFieldNumber.h"

#include "BCH\_coder.h"

Від тепер можна користуватись в коді класами: GaloisFieldNumber, BCH\_Codec.

Конструктор GaloisFieldNumber, приймає два числа: перше цілу ступінь 2 для поля Галуа; другий – десяткове ціле число.

Конструктор BCH\_Codec, приймає на вході два числа: перше цілу ступінь 2 для поля Галуа; друге кількість помилок що слід виправити. Для операції encode(), на вхід треба передати vector<unsigned char>, на вихіді цей метод повертає структуру EncodedMessage: що має наступні поля:

* std::vector<unsigned char> encodedMessage;
* unsigned int totalLengthInBites = 0;
* unsigned int originalMessageLenghtInBites = 0;

Саме таку структуру слід передавати на вхід методу decode(), котрий повертає vector<unsigned char> з декодованим повідомленням. Якщо сталась помилка в декодуванні, то вектор буде пустим.

# 

# ВИСНОВКИ

В результаті практики було отримано пакет кодеку, який реалізує БЧХ кодування та декодування повідомлень, як без помилок в повідомленні для декодування, так і з помилками. Пакет коректо знаходить локатори синдромів (помилок) та успішно виправляє похибки якщо вони підходять під умови декодування.

Через те що пакет написано на мові с++11, то він працює швидко і використовує незначну кількість оперативної пам’яті. Через вид реалізації пакету, кінцевий користувач (програміст) може легко розпаралелити роботу кодеку на декілька потоків, в залежності від бажаної кількісті та архітектури комп’ютера. В пакеті не використаються рішення, що не гарантуються стандартом, тому його можна компілювати під будь які платформи що дотримуються стандарту.

Також через вид дистрибуції пакету, програміст може модифікувати код на свій смак в залежності від того, що бажає отримати. Наприклад, в пакеті не пітримуются числа більші ніж unsigned long, хочу для деяких платформ можна легко виправити це до unsigned long long, що в залежності від платформи можуть бути роміром 8 байт, або 256 бітів, тим самим використати всю потужність кодеку на дійсно великих розмірах простору Галуа (але не біль 28, тому що таблиці незворотних поліномів до поля забиті лише до 28 включно).