МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Университет ИТМО
Факультет систем управления и робототехники

ЛАБОТАТОРНАЯ РАБОТА № 1

по дисциплине «Практическая линейная алгебра»

Выполнила:
Студентка группы R3281
Троицкая Тамара Андреевна
Преподаватель:
Перегудин Алексей Алексеевич

Санкт-Петербург, 2023

Содержание

Intro

Приветствую тебя, уважаемый читатель моего отчёта. Он обещает быть обширным, так что перед началом я посоветовала бы тебе заварить кофе или чего покрепче.

Код для лабы я писала на Си, поэтому написала вручную все структуры и функции, потребовавшиеся для работы с матрицами и векторами. Я постаралась написать комментарии к коду максимально подробно, чтобы всё было понятно даже человеку, никогда не писавшему на Си. Можете просто читать комментарии перед функциями, чтобы понять, что они делают, но реализацию я тоже приложу.

Я создала 2 проекта. Первый, Hills_cypher, реализует вычисление Task 1 и Task 2. Второй, Hammings_code, реализует Task 3.

Можете ориентироваться в отчёте, переходя по ссылкам в содержании. Под каждым из заданий перечислены файлы, по которым распределено решение. Если кликнуть на название файла, вы перейдёте к описанию реализованных там функций.

Task 1

Начнём стандартно с задачи №1, в которой нужно было реализовать шифр Хилла. Говоря коротко, нужно было:

1. Придумать квадратную матрицу-ключ
2. Перевести сообщение из 12 букв в массив из 12 чисел -- номеров этих букв в заданном “алфавите”
3. Разбить массив этих чисел в массив векторов такого же размера, как матрица-ключ
4. Поочерёдно умножать матрицу-ключ на эти векторы, формируя массив векторов результата
5. Сконкатенировать, то есть склеить все векторы в один массив чисел
6. Перевести этот массив чисел обратно в буквы

structs.h

Теперь, когда идея понятна, перейдём к коду. Файл structs.h. В нём реализованы структуры и объявлены функции, реализованные в файле structs.c. Эти функции можно будет использовать в любом файле, в начале которого написано #include “structs.h”. Это именно те вспомогательные функции для работы с матрицами и векторами, о которых я говорила вначале. Скоро рассмотрим действие каждой из них более подробно.

Я написала структуры матрицы и вектора, чтобы можно было в явном виде задавать и использовать размеры матрицы и вектора. Это было бы труднее и не так безопасно, если бы мы использовали просто указатели. Также при моём подходе можно создавать указатель на матрицу и массив векторов, не путаясь в количестве звёздочек. Также я называла функции так, чтоб их название максимально понятно показывало, что делает данная функция.

```c
#pragma once

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>

#define message_len 12
#define abc_len 30

extern char abc[];

// вектор длины n
struct vec {
    size_t n;
    size_t* v;
};

// квадратная матрица n * n
struct square_matrix {
    size_t n;
    size_t** arr;
};

void print(size_t s);
void print_int64_t(int64_t s);
void print_string(char* s);
void print_vec(struct vec* v);
void print_matr(struct square_matrix* M);

struct square_matrix matr2x2(size_t arr[2][2]);
struct square_matrix matr3x3(size_t arr[3][3]);
struct square_matrix matr4x4(size_t arr[4][4]);

struct vec matr_mul_vec(struct square_matrix *M, struct vec *V);

int64_t det(struct square_matrix* M);

int64_t mod_abc_len(int64_t a);

struct square_matrix minor(struct square_matrix *M, size_t x, size_t y);
struct square_matrix invert(struct square_matrix* M);

```

structs.c

Перейдём к рассмотрению реализации функций. Файл structs.c. В начале файла -- массив используемых символов алфавита. Далее первые 5 функций нужны для того, чтобы выводить в консоль разные типы данных. Это пригодится для отладки и просмотра результата.

```c
#include "headers/structs.h"

char abc[] = "абвгдежзийклмнопрстуфхцчшыьэюя";

// выводит в консоль число типа size_t
void print(size_t s) {
    printf("%zu ", s);
}

// выводит в консоль число типа int64_t
void print_int64_t(int64_t s) {
    printf("%" PRId64 " ", s);
}

// выводит в консоль строку длины 12
void print_string(char* s) {
    for (size_t i = 0; i < message_len; i++) {
        printf("%c", *(s + i));
    }
    printf("\n\n");
}

// выводит в консоль координаты вектора
void print_vec(struct vec* v) {
    for (size_t i = 0; i < v->n; i++) {
        printf("%zu ", *(v->v + i));
    }
    printf("\n");
}

// выводит в консоль матрицу
void print_matr(struct square_matrix* M) {
    size_t** arr = M->arr;
    size_t n = M->n;
    for (size_t i = 0; i < n; i++) {
        print_vec(&((struct vec) { n, *(M->arr + i) }));
    }
    printf("\n");
}
```

Две функции для работы в кольце (в общем, чтоб искать что-то по модулю)

```c
// возвращает число по модулю, friendly к отрицательным числам
int64_t mod_abc_len(int64_t a) {
    if (a >= 0) return a % abc_len;
    while (a < 0) a += abc_len;
    return a % abc_len;
}

// возвращает число, обратное по модулю размера алфавита
int64_t inv_mod(int64_t a) {
    for (size_t i = 0; i < abc_len; i++) {
        if (mod_abc_len(a * i) == 1)
            return i;
    }
    return -1;
}
```

Функции для упрощённой инициализации матриц с помощью двумерного массива:

```c
// создаёт двойной указатель по двумерному массиву и оборачивает в структуру
struct square_matrix matr2x2(size_t arr[2][2]) {
    size_t n = 2;
    size_t** res = (size_t**)malloc(n * sizeof(size_t*));
    for (size_t i = 0; i < n; i++)
        res[i] = (size_t*)malloc(n * sizeof(size_t));
    for (size_t i = 0; i < n; i++) {
        for (size_t j = 0; j < n; j++) {
            res[i][j] = arr[i][j];
        }
    }
    return (struct square_matrix) { n, res };
}

// создаёт двойной указатель по двумерному массиву и оборачивает в структуру
struct square_matrix matr3x3(size_t arr[3][3]) {
    size_t n = 3;
    size_t** res = (size_t**)malloc(n * sizeof(size_t*));
    for (size_t i = 0; i < n; i++)
        res[i] = (size_t*)malloc(n * sizeof(size_t));
    for (size_t i = 0; i < n; i++) {
        for (size_t j = 0; j < n; j++) {
            res[i][j] = arr[i][j];
        }
    }
    return (struct square_matrix) { n, res };
}

// создаёт двойной указатель по двумерному массиву и оборачивает в структуру
struct square_matrix matr4x4(size_t arr[4][4]) {
    size_t n = 4;
    size_t** res = (size_t**)malloc(n * sizeof(size_t*));
    for (size_t i = 0; i < n; i++)
        res[i] = (size_t*)malloc(n * sizeof(size_t));
    for (size_t i = 0; i < n; i++) {
        for (size_t j = 0; j < n; j++) {
            res[i][j] = arr[i][j];
        }
    }
    return (struct square_matrix) { n, res };
}
```

```c
// умножает квадратную матрицу на вектор
struct vec matr_mul_vec(struct square_matrix *M, struct vec *V) {
    size_t n = V->n;
    // валидация размеров
    if (M->n != n) {
        printf("ЭТУ МАТРИЦУ И ВЕКТОР НЕЛЬЗЯ ПЕРЕМНОЖАТЬ\n");
        printf("matr %zu vec %zu\n", M->n, n);
        return (struct vec) { 0, NULL };
    }
    size_t** m = M->arr, * v = V->v;
    size_t* res = (size_t*)malloc(n * sizeof(size_t));
    for (size_t i = 0; i < n; i++) {
        res[i] = 0;
        for (size_t j = 0; j < n; j++) {
            res[i] += mod_abc_len((m[i][j] * v[j]));
        }
        res[i] = mod_abc_len(res[i]);
    }
    return (struct vec) { n, res };
}
```

Возвращает минор

```c
// возвращает минор матрицы по заданным индексам
struct square_matrix minor(struct square_matrix *M, size_t x, size_t y) {
    size_t** a = M->arr;
    size_t n = M->n;
    size_t** res = (size_t**)malloc((n - 1) * sizeof(size_t*));
    for (size_t i = 0; i < n; i++)
        res[i] = (size_t*)malloc((n - 1) * sizeof(size_t));
    size_t i_m = 0, j_m;
    for (size_t i = 0; i < n; i++) {
        if (i == x) continue;
        j_m = 0;
        for (size_t j = 0; j < n; j++) {
            if (j != y && i_m < n && j_m < n) {
                res[i_m][j_m] = a[i][j];
                j_m++;
            }
        }
        if (i != x) {
            i_m++;
        }
    }
    return (struct square_matrix) { n - 1, res };
}
```

Функции для подсчёта определителя (последняя нужна для укороченного вызова первых трёх)

```c
// определитель матрицы 2x2
int64_t det2x2(struct square_matrix *M) {
    size_t** a = M->arr;
    return (int64_t)a[0][0] * a[1][1] - a[0][1] * a[1][0];
}

// определитель матрицы 3x3
int64_t det3x3(struct square_matrix *M) {
    size_t** a = M->arr;
    return (int64_t)a[0][0] * a[1][1] * a[2][2]
        + a[0][1] * a[1][2] * a[2][0]
        + a[0][2] * a[1][0] * a[2][1]
        - a[0][2] * a[1][1] * a[2][0]
        - a[0][1] * a[1][0] * a[2][2]
        - a[0][0] * a[1][2] * a[2][1];
}

// определитель матрицы 4x4, посчитанный через разложение по первой строке
int64_t det4x4(struct square_matrix* M) {
    size_t** a = M->arr;
    size_t n = 4;
    int64_t res = 0, coef = 1;
    for (size_t i = 0; i < n; i++) {
        struct square_matrix cur_minor = minor(M, 0, i);
        res += coef * a[0][i] * det3x3(&cur_minor);
        coef *= -1;
    }
    return res;
}

// определитель матриц от 1x1 до 4x4
int64_t det(struct square_matrix* M) {
    if (M->n == 1) return (M->arr)[0][0];
    if (M->n == 2) return det2x2(M);
    if (M->n == 3) return det3x3(M);
    if (M->n == 4) return det4x4(M);
    printf("ОШИБКА ОПРЕДЕЛИТЕЛЯ\n");
    return -1;
}
```

Напомню факт из матричной алгебры. Чтобы посчитать матрицу, обратную данной, надо сначала построить матрицу, состоящую из определителей её миноров, затем сделать из неё матрицу алгебраических дополнений, домножив нужные элементы на -1, после транспонировать её и полученную матрицу домножить на число, обратное дискриминанту, не забывая, что мы в кольце вычетов по модулю:

```c
// обращает матрицу (используется для дешифровки)
struct square_matrix invert(struct square_matrix* M) {
    size_t n = M->n;
    int64_t** res = (int64_t**)malloc(n * sizeof(int64_t*)), coef;
    for (size_t i = 0; i < n; i++)
        res[i] = (int64_t*)malloc(n * sizeof(int64_t));
    // adjusted
    for (size_t i = 0; i < n; i++) {
        for (size_t j = 0; j < n; j++) {
            if (i % 2 != j % 2) {
                coef = -1;
            }
            else coef = 1;
            struct square_matrix minorr = minor(M, i, j);
            res[i][j] = mod_abc_len(det(&minorr) * coef);
        }
    }
    // transpose
    res = transpose(res, n);
    // multiply to inv to det
    int64_t determ = inv_mod(mod_abc_len(det(M)));
    if (determ == -1) {
        printf("МАТРИЦА НЕОБРАТИМА\n");
        return (struct square_matrix) { 0, NULL };
    }
    for (size_t i = 0; i < n; i++) {
        for (size_t j = 0; j < n; j++) {
            res[i][j] = mod_abc_len(determ * res[i][j]);
        }
    }
    return (struct square_matrix) { n, res };
}

```

Итак, мы рассмотрели все вспомогательные функции работы с матрицами и векторами. Перейдём непосредственно к шифрованию.

Task1.c

В начале мы импортировали все написанные нами функции для работы с матрицами. Потом импортировали время, оно нам ещё понадобится.

Для начала нам понадобятся функции, превращающие букву в её номер в алфавите и наоборот, номер в букву. Вот эти две функции:

```c
#include "headers/structs.h"
#include <time.h>

// возвращает букву по номеру
char get_letter(size_t s) {
    if (s >= 0 && s < abc_len)
        return abc[s];
    return -1;
}

// возвращает номер буквы в заданном алфавите
size_t get_number(char letter) {
    for (size_t i = 0; i < 30; i++) {
        if (letter == abc[i]) {
            return i;
        }
    }
    printf("there is no such letter\n");
    return -1;
}
```

Массив букв одновременно переводит в числа и разбивает на несколько векторов (пункты 2 и 3 исходного плана со страницы 2):

```c
// превратить массив char в массив векторов нужной длины
struct vec* vectors(char* s, size_t vec_size, size_t num_of_vec) {
    struct vec* vectors = (struct vec*)malloc(num_of_vec * sizeof(struct vec));
    for (size_t i = 0; i < num_of_vec; i++) {
        size_t* tmp = (size_t*)malloc(vec_size * sizeof(size_t));
        for (size_t j = 0; j < vec_size; j++) {
            tmp[j] = get_number(s[i * vec_size + j]);
        }
        *(vectors + i) = (struct vec){ vec_size, tmp };
    }
    return vectors;
}
```

Выполняет действие, обратное к действию предыдущей функции (5 и 6 пункт):

```c
// перевести массив векторов в массив букв
char* nums_to_string(struct vec* vectors, size_t vec_size, size_t num_of_vec) {
    char* res = (char*)malloc(vec_size * sizeof(char));
    for (size_t i = 0; i < num_of_vec; i++) {
        for (size_t j = 0; j < vec_size; j++) {
            *(res + i * vec_size + j) = get_letter((vectors[i].v)[j]);
        }
    }
    return res;
}
```

cipher -- полностью выполняет шифрование по плану на стр 2 (строки 50-53 выполняют п.4)

Вызывает в нужном порядке описанные выше функции, объединяет их в единую логику.

decipher выполняет дешифрование. Как вы можете заметить, в ней происходит обращение матрицы-ключа и вызов функции cipher с полученной матрицей.

```
46  // зашифровывает слово методом Хилла по заданной матрице-ключу
47  char* cipher(struct square_matrix *m, char* s) {
48      size_t vec_size = m->n, num_of_vec = message_len / vec_size;
49      struct vec* vecs = vectors(s, vec_size, num_of_vec);
50      struct vec* multipied = (struct vec*)malloc(num_of_vec * sizeof(struct vec));
51      for (size_t i = 0; i < num_of_vec; i++) {
52          *(multipied + i) = matr_mul_vec(m, vecs + i);
53      }
54      return nums_to_string(multipied, vec_size, num_of_vec);
55  }
56
57  // обращает матрицу-ключ и дешифрует сообщение
58  char* decipher(struct square_matrix *m, char* ciphr) {
59      struct square_matrix inv = invert(m);
60      return cipher(&inv, ciphr);
61  }
```

Также по заданию нужно было добавить 3 опечатки в каждую из зашифрованных строк.

rand() % 30 даёт целое число в диапазоне [0; 29]

Эта функция изменяет исходное сообщение.

```
63  // добавить 3 опечатки в сообщение
64  char* tree_typos(char* message) {
65      size_t
66          i1 = rand() % message_len,
67          i2 = rand() % message_len,
68          i3 = rand() % message_len;
69      *(message + i1) = get_letter(rand() % abc_len);
70      *(message + i2) = get_letter(rand() % abc_len);
71      *(message + i3) = get_letter(rand() % abc_len);
72      return message;
73  }
```

И вот мы наконец добрались до самой главной функции, которая запускает весь проект.

Чтобы рандом был ещё рандомнее, мы делаем srand(clock());

clock() (именно ради него мы подключали <time.h>) возвращает текущее время в некоторых особых единицах, а srand как-то на нём основывает все псевдорандомные числа, генерящиеся в ходе работы программы.

В строке 80 записана шифруемая строка из 12 символов.

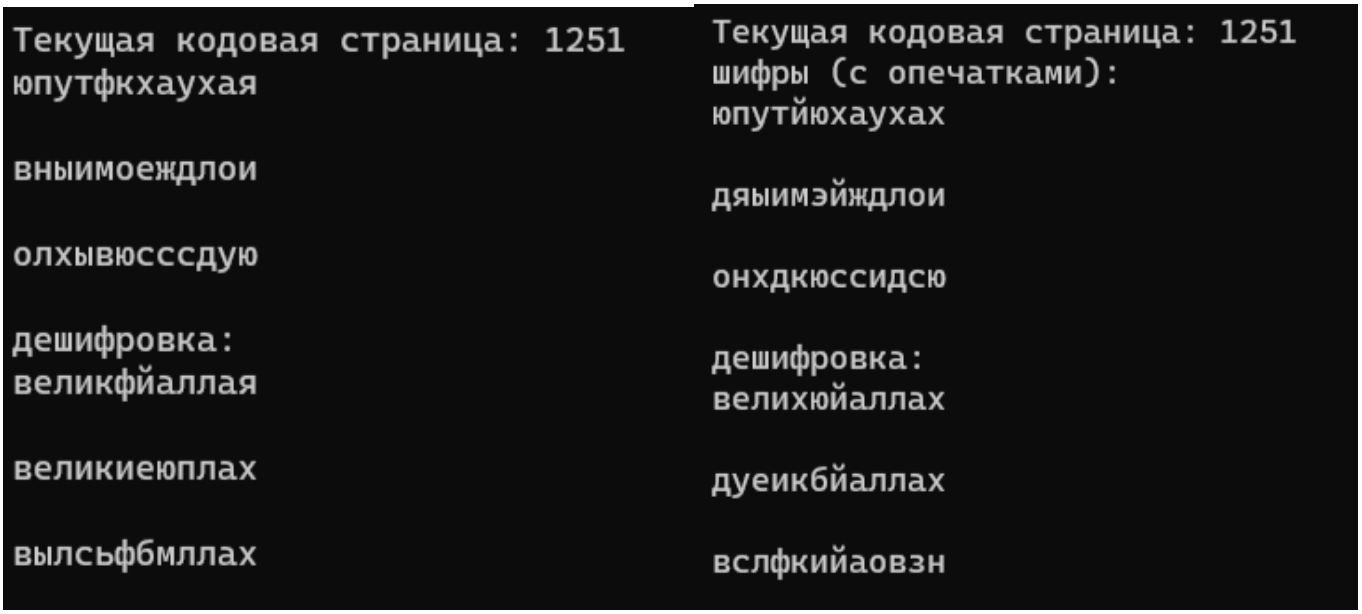
Строки 82-97 задают три матрицы-ключа, с помощью которых производится шифрование. Все их элементы взяты по модулю 30 (30 -- количество букв в заданном алфавите), поэтому они находятся в диапазоне от 0 до 29. Так как они треугольны, то их определитель -- произведение элементов на главной диагонали, очевидно, что 30 и эти определители являются взаимно простыми.

В строках 100-103 мы зашифровали сообщение тремя путями (матрицами m2, m3 и m4), получив массивы символов.

Далее в каждый из массивов этих символов мы добавили по 3 опечатки, то есть поменяли псевдорандомные 3 буквы в зашифрованных сообщениях. (105-108)

И наконец в строках 115-117 мы дешифруем эти сообщения и выводим результат на экран.

```
74  int main() {
75      // кодировка
76      system("chcp 1251");
77      srand(clock());
78      // шифруемое сообщение
79      char* secret = "великийаллах";
80      // матрицы-ключи
81      size_t a3[3][3] = {
82          {1,0,0},
83          {4,7,0},
84          {7,8,11}
85      };
86      size_t a4[4][4] = {
87          {7, 0, 0, 0},
88          {3, 11, 0, 0},
89          {-1, 0, 13, 0},
90          {8, 21, 1, 29}
91      };
92      struct square_matrix
93          m2 = matr2x2((size_t[2][2]) { {29, 0}, {20, 1} }),
94          m3 = matr3x3(a3),
95          m4 = matr4x4(a4);
96      // зашифровать тремя разными матрицами
97      char *c2 = cipher(&m2, secret),
98          *c3 = cipher(&m3, secret),
99          *c4 = cipher(&m4, secret);
100     // добавить по 3 опечатки в каждое сообщение
101     tree_typos(c2);
102     tree_typos(c3);
103     tree_typos(c4);
104     // напечатать зашифрованные строки с опечатками
105     printf("шифры (с опечатками):\n");
106     print_string(c2);
107     print_string(c3);
108     print_string(c4);
109     // напечатать результат дешифровки строк с опечатками
110     printf("дешифровка:\n");
111     print_string(decipher(&m2, c2));
112     print_string(decipher(&m3, c3));
113     print_string(decipher(&m4, c4));
114
115     return 0;
116 }
```



Анализ результатов работы программы

Можно заметить, что 3 опечатки в зашифрованном сообщении могут повлечь за собой большее количество неправильных символов во втором. Почему это происходит? Ответ на вопрос становится очевидным если вспомнить, как вычисляется умножение матрицы на вектор. Строка матрицы “умножается” на столбец вектора. То есть если вектор неправильный, это порождает ошибку в худшем количестве векторов, равном количеству строк матрицы. В некоторых случаях буква может оказаться неошибочной, особенно при небольшом алфавите, так как все числа берутся по модулю числа символов алфавита.

Вывод

Шифр Хилла уязвим при изменениях зашифрованного сообщения. То есть если мы немного поменяем шифр, это довольно сильно скажется на расшифрованном сообщении.

Интересно, дочитал ли кто-нибудь до этой строчки...

Task 2

Второе задание. Мы хакеры. У какого-то пользователя было 2 сообщения. Он их зашифровал одной матрицей-ключом. Нам в руки попало 2 зашифрованных сообщения и расшифровка первого сообщения. Матрица-ключ нам неизвестна, и по сути её нам надо найти. Точнее сказать, нам нужно найти матрицу, обратную к матрице-ключу, чтобы с помощью её расшифровать потом второе сообщение пользователя. То есть наша задача -- расшифровать первое сообщение пользователя.

Для второй задачи я создала другой файл (Task2.c), и теперь чтобы использовать функции, реализованные в файле Task1.c, надо создать заголовочный файл и занести в него все функции, которые мы собираемся использовать. (hill.h)

```c
#pragma once

#include "structs.h"

char get_letter(size_t s);
size_t get_number(char letter);

char* nums_to_string(struct vec* vectors, size_t vec_size, size_t num_of_vec);
struct vec* vectors(char* s, size_t vec_size, size_t num_of_vec);

char* cipher(struct square_matrix* m, char* s);
char* decipher(struct square_matrix* m, char* cyphr);

```

Вторая функция создаёт матрицу-ключ. Здесь это действие не автоматизировано, но так как мы вынесли это в отдельную функцию, в дальнейшем можно будет заменить алгоритм этого более низкого уровня, оставив без изменений высокий уровень.

```c
#include "headers/hill.h"

// проверяет, равны ли векторы
size_t equal_vec(struct vec cyph, struct vec orig) {
    if (cyph.n != orig.n) return 0;
    for (size_t i = 0; i < cyph.n; i++) {
        if (cyph.v[i] != orig.v[i]) return 0;
    }
    return 1;
}

// создаёт матрицу-ключ 2x2, обратную к которой хотим "угадать"
struct square_matrix matr_gen() {
    struct square_matrix
        m = matr2x2((size_t[2][2]) {
            {17, 3},
            { 2, 1 }});
    return m;
}
```

Эта функция перебирает все 810 тысяч вариантов матрицы (по 30 вариантов на каждый элемент матрицы) и потом проверяет, верный ли результат она даёт при умножении её на каждый из векторов массива:

```c
// перебором находит матрицу
struct square_matrix hack_key_dumb_edition(struct vec* cyph, struct vec* orig) {
    size_t flag = 1;
    for (size_t i1 = 0; i1 < abc_len; i1++) {
        for (size_t i2 = 0; i2 < abc_len; i2++) {
            for (size_t i3 = 0; i3 < abc_len; i3++) {
                for (size_t i4 = 0; i4 < abc_len; i4++) {
                    flag = 1;
                    // перебираем все пары векторов
                    struct square_matrix mbres = matr2x2((size_t[2][2]) { {i1, i2}, { i3, i4 } });
                    for (size_t i5 = 0; i5 < message_len / 2; i5++) {
                        struct vec mul = matr_mul_vec(&mbres, (cyph + i5));
                        if (!equal_vec(mul, *(orig + i5))) flag = 0;
                    }
                    if (flag)
                        return mbres;
                }
            }
        }
    }
    printf("NO RESULT\n");
    return (struct square_matrix) { 0, NULL };
}
```

Функция main. Думаю, её действие довольно очевидно из комментариев.

```c
int main() {
    // кодировка
    system("chcp 1251");
    size_t vec_size = 2;
    // дана первая расшифровка, найти вторую
    char* s1 = "недвенадцать",
        *s2 = "троицкаятома";
    struct square_matrix m = matr_gen();
    // даны два зашифрованных сообщения
    char* c1 = cipher(&m, s1),
        *c2 = cipher(&m, s2);

    // расшифровка и зашифровка первого сообщения в удобном виде
    struct vec* cyf1 = vectors(s1, vec_size, message_len / vec_size),
        * decyf1 = vectors(c1, vec_size, message_len / vec_size);

    struct square_matrix inv = hack_key_dumb_edition(decyf1, cyf1);
    print_matr(&inv);
    print_string(cipher(&inv, c2));
    return 0;
}

```

```
Текущая кодовая страница: 1251
11 27
8 7

троицкаятома
```

Вывод

В данной задаче даже прямой перебор работает вполне быстро. Взлом оказался очень простым. Получается, чтобы эффективно пользоваться шифром Хилла, под каждое сообщение нужно придумывать новый ключ, иначе при утечке одного из исходных сообщений можно будет с лёгкостью раскрыть все.

Task 3

Третье задание уже принципиально другое. Код Хэмминга. Хотя некоторые из функций нам пригодятся из прошлой задач, импортировать заголовочный файл structs.h я не стала, чтобы не импортировать ненужные нам структуры и функции, в которых можно запутаться, и чтобы избежать конфликта имён некоторых глобальных переменных. Для этой задачи нам нужна будет не квадратная, а произвольная матрица. Также я создала структуру map для того, чтобы хранить пару код - буква. Строго говоря, структура map реализует пару, а массив из таких элементов уже можно было бы называть map.

```c
#pragma once

#include <stdio.h>
#include <stdlib.h> // malloc
#include <time.h>

#define abc_size 32
#define code_len 5

struct matrix {
    size_t n, m;
    size_t** arr;
};

struct vec {
    size_t n;
    size_t* v;
};

struct map {
    size_t* code;
    char letter;
};

void print_str(size_t* s, size_t n);
void print_string(char* s);
void print_vec(struct vec* v);

size_t equals(size_t* a, size_t* b);
struct map map(char* c, char letter);

struct vec nonsquare_matr_mul_vec(struct matrix* M, struct vec* V);
struct matrix transpose(struct matrix m);

```

structs.c

Функции, аналогичные функциям из предыдущего задания.

```c
#include "headers/structs.h"

// вывести в консоль массив size_t длины n
void print_str(size_t* s, size_t n) {
    for (size_t i = 0; i < n; i++) {
        printf("%zu", *(s + i));
    }
    printf(" ");
}

// вывести в консоль массив символов длины 4
void print_string(char* s) {
    for (size_t i = 0; i < 4; i++) {
        printf("%c", *(s + i));
    }
    printf("\n\n");
}

// вывести в консоль значения координат вектора (struct vec)
void print_vec(struct vec* v) {
    for (size_t i = 0; i < v->n; i++) {
        printf("%zu ", *(v->v + i));
    }
    printf("\n");
}

size_t equals(size_t* a, size_t* b) {
    for (size_t i = 0; i < 5; i++) {
        if (a[i] != b[i]) return 0;
    }
    return 1;
}
```

Более общий случай функций для работы с произвольными матрицами.

```c
// умножает по модулю 2 неквадратную матрицу на вектор
struct vec nonsquare_matr_mul_vec(struct matrix* M, struct vec* V) {
    size_t res_n = M->n;
    size_t* res = (size_t*)malloc(res_n * sizeof(size_t));
    // проверка совместимости размеров матрицы и вектора
    size_t n = V->n;
    if (M->m != n) {
        printf("ЭТУ МАТРИЦУ И ВЕКТОР НЕЛЬЗЯ ПЕРЕМНОЖАТЬ\n");
        printf("matr %zu vec %zu\n", M->m, n);
        return (struct vec) { 0, NULL };
    }
    size_t** m = M->arr, * v = V->v;
    for (size_t i = 0; i < res_n; i++) {
        res[i] = 0;
        for (size_t j = 0; j < n; j++) {
            res[i] += m[i][j] * v[j];
        }
        res[i] = (res[i]) % 2;
    }
    return (struct vec) { res_n, res };
}

// возвращает транспонированную матрицу
struct matrix transpose(struct matrix m) {
    // выделить память
    size_t** res = (size_t**)malloc(m.m * sizeof(size_t*));
    for (size_t i = 0; i < m.n; i++)
        res[i] = (size_t*)malloc(m.n * sizeof(size_t));
    // найти транспонированную матрицу
    for (size_t i = 0; i < m.m; i++) {
        for (size_t j = 0; j < m.n; j++) {
            res[i][j] = m.arr[j][i];
        }
    }
    return (struct matrix) { m.m, m.n, res };
}
```

Принципиально новая функция. Она создаёт экземпляр структуры map упрощённым образом.

```c
// функция для более быстрого создания пары код - буква
struct map map(char* c, char letter) {
    size_t* code = (size_t*)malloc(sizeof(size_t) * code_len);
    for (size_t i = 0; i < code_len; i++) {
        if (c[i] == '0') {
            code[i] = 0;
        }
        else if (c[i] == '1')
            code[i] = 1;
    }
    return (struct map) { code, letter };
}
```

Task3.c

В начале файла мы создаём map, задающий пятизначный код для каждой буквы алфавита.

```c
#include "headers/structs.h"

struct map* abc;

// создать массив пар ключ-значение для буквы и её двоичного кода
void abc_init(void) {
    abc = (struct map*)malloc(abc_size * sizeof(struct map));
    size_t i = 0;
    abc[i++] = map("00000", 'а');
    abc[i++] = map("00001", 'б');
    abc[i++] = map("00010", 'в');
    abc[i++] = map("00011", 'г');
    abc[i++] = map("00100", 'д');
    abc[i++] = map("00101", 'е');
    abc[i++] = map("00110", 'ж');
    abc[i++] = map("00111", 'з');
    abc[i++] = map("01000", 'и');
    abc[i++] = map("01001", 'й');
    abc[i++] = map("01010", 'к');
    abc[i++] = map("01011", 'л');
    abc[i++] = map("01100", 'м');
    abc[i++] = map("01101", 'н');
    abc[i++] = map("01110", 'о');
    abc[i++] = map("01111", 'п');
    abc[i++] = map("10000", 'р');
    abc[i++] = map("10001", 'с');
    abc[i++] = map("10010", 'т');
    abc[i++] = map("10011", 'у');
    abc[i++] = map("10100", 'ф');
    abc[i++] = map("10101", 'х');
    abc[i++] = map("10110", 'ц');
    abc[i++] = map("10111", 'ч');
    abc[i++] = map("11000", 'ш');
    abc[i++] = map("11001", 'щ');
    abc[i++] = map("11010", 'ъ');
    abc[i++] = map("11011", 'ы');
    abc[i++] = map("11100", 'ь');
    abc[i++] = map("11101", 'э');
    abc[i++] = map("11110", 'ю');
    abc[i++] = map("11111", 'я');
}
```

Далее реализованы две функции с противоположным действием:

```c
// получить двоичный код из 5 символов для буквы
size_t* get_letter_code(char letter) {
    for (size_t i = 0; i < abc_size; i++) {
        if (abc[i].letter == letter) {
            return abc[i].code;
        }
    }
    printf("THERE'S NO SUCH LETTER\n");
    return NULL;
}

// получить букву из массива 5 двоичных символов
char get_letter(size_t* code) {
    for (size_t i = 0; i < abc_size; i++) {
        if (equals(code, abc[i].code))
            return abc[i].letter;
    }
    printf("ТАКОЙ БУКВЫ НЕТ\n");
    return 0;
}
```

Далее функция, которая из массива чисел создаёт массив векторов заданной длины, заданного количества.

```c
// функция для упрощённого задания массива векторов
struct vec* vecs(size_t* s, size_t vec_size, size_t num_of_vec) {
    struct vec* vectors = (struct vec*)malloc(num_of_vec * sizeof(struct vec));
    for (size_t i = 0; i < num_of_vec; i++) {
        size_t* tmp = (size_t*)malloc(vec_size * sizeof(size_t));
        for (size_t j = 0; j < vec_size; j++) {
            tmp[j] = s[i * vec_size + j];
        }
        *(vectors + i) = (struct vec){ vec_size, tmp };
    }
    return vectors;
}
```

```c
// по заданному массиву двоичных символов возвращает тот же массив с 1 опечаткой
size_t* typo(size_t* message) {
    size_t message_len = 35;
    size_t i1 = rand() % message_len;
    printf("typo index: %zu\n", i1);
    message[i1] = (message[i1] + 1) % 2;
    return message;
}
```

И вот функция, которая кодирует слово методом Хэмминга:

1. Превращает каждую из 4х букв в массив из 5 двоичных символов, получаем массив из 20 двочиных символов
2. Разбивает массив из 20 символов в 5 векторов по 4 символа
3. Умножает каждый из 5ти векторов на матрицу G (здесь объяснено, откуда она взялась). Итак, получаем 5 векторов длины 7.
4. Склеиваем все векторы в один и получаем массив из 35 двоичных знаков.

```
// кодирует слово из четырёх букв, возвращает строку из 35 двоичных символов
size_t* encoding(char* word) {
    size_t mes_len = 4;
    // массив из чисел из букв
    size_t* nums = (size_t*)malloc(sizeof(size_t) * mes_len * code_len);
    for (size_t i = 0; i < mes_len; i++) {
        size_t* tmp = get_letter_code(word[i]);
        for (size_t j = 0; j < code_len; j++) {
            nums[code_len * i + j] = tmp[j];
        }
    }
    // разбиваем на векторы длины 4
    struct vec* orig_vecs = vecs(nums, 4, 5);
    // задаём матрицу G
    size_t matr_G[7][4] = {
        {1, 1, 0, 1},
        {1, 0, 1, 1},
        {1, 0, 0, 0},
        {0, 1, 1, 1},
        {0, 1, 0, 0},
        {0, 0, 1, 0},
        {0, 0, 0, 1}
    };
    // выделяем память, превращаем массив в двойной указатель, присываиваем адреса
    size_t** G = (size_t**)malloc(7 * sizeof(size_t*));
    for (size_t i = 0; i < 7; i++)
        G[i] = (size_t*)malloc(4 * sizeof(size_t));
    for (size_t i = 0; i < 7; i++) {
        for (size_t j = 0; j < 4; j++) {
            G[i][j] = matr_G[i][j];
        }
    }
    // оборачиваем двойной указатель в структуру
    struct matrix key = (struct matrix){ 7, 4, G };
    // умножаем матрицу на векторы длины 4 и получаем векторы длины 7
    struct vec* multiplied = (struct vec*)malloc(5 * sizeof(struct vec));
    for (size_t i = 0; i < 5; i++) {
        *(multiplied + i) = nonsquare_matr_mul_vec(&key, orig_vecs + i);
    }
    // печатаем исходные и закодированные векторы
    printf("vec:\n");
    for (size_t i = 0; i < 5; i++) {
        print_vec(&orig_vecs[i]);
        print_vec(&multiplied[i]);
    }
    printf("\n");
    // склеиваем векторы в один массив из 35 двоичных символов
    size_t* encoded = (size_t*)malloc(35 * sizeof(size_t));
    for (size_t i = 0; i < 5; i++) {
        for (size_t j = 0; j < 7; j++) {
            encoded[i * 7 + j] = (multiplied[i]).v[j];
        }
    }
    return encoded;
}
```

```c
// перевести трёхзначный двоичный код, записанный задом наперёд, в число
size_t number(size_t* binary) {
    if (binary[0] == 0 && binary[1] == 0 && binary[2] == 0)
        return 0;
    if (binary[0] == 1 && binary[1] == 1 && binary[2] == 1)
        return 7;
    if (binary[0] == 0 && binary[1] == 1 && binary[2] == 1)
        return 6;
    if (binary[0] == 1 && binary[1] == 0 && binary[2] == 1)
        return 5;
    if (binary[0] == 0 && binary[1] == 0 && binary[2] == 1)
        return 4;
    if (binary[0] == 1 && binary[1] == 1 && binary[2] == 0)
        return 3;
    if (binary[0] == 0 && binary[1] == 1 && binary[2] == 0)
        return 2;
    if (binary[0] == 1 && binary[1] == 0 && binary[2] == 0)
        return 1;
}
```

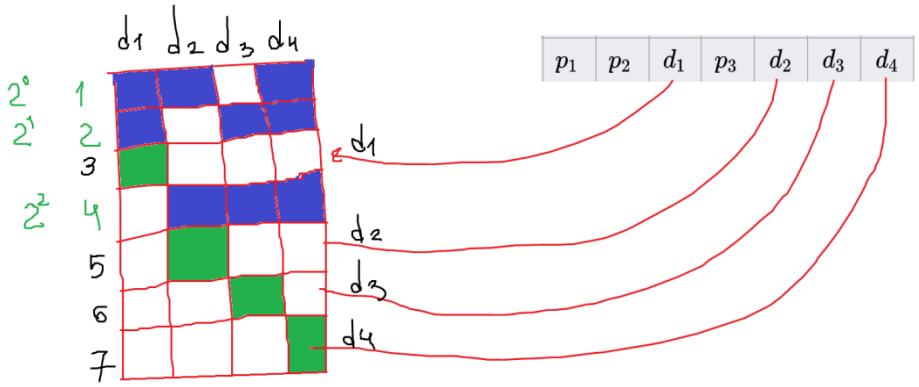
```c
// проверяет, есть ли опечатки. возвращает векторы без опечаток
struct vec* check_parity_bits(size_t* encoded) {
    // матрица, показывающая, за какие биты отвечает какой бит чётности
    size_t matr_H[3][7] = {
        {1, 0, 1, 0, 1, 0, 1},
        {0, 1, 1, 0, 0, 1, 1},
        {0, 0, 0, 1, 1, 1, 1}
    };
    size_t** h = (size_t**)malloc(3 * sizeof(size_t*));
    for (size_t i = 0; i < 3; i++)
        h[i] = (size_t*)malloc(7 * sizeof(size_t));
    for (size_t i = 0; i < 3; i++) {
        for (size_t j = 0; j < 7; j++) {
            h[i][j] = matr_H[i][j];
        }
    }
    struct matrix H = (struct matrix){ 3, 7, h };

    size_t num_of_vec = 5, vec_size = 7;
    // если умножить проверочную матрицу на векторы, должны в идеале получиться все нули
    struct vec* decipher = (struct vec*)malloc(num_of_vec * sizeof(struct vec));
    // надо сначала разбить encoded на векторы длины 7
    struct vec* vectors = (struct vec*)malloc(num_of_vec * sizeof(struct vec));
    for (size_t i = 0; i < num_of_vec; i++) {
        size_t* tmp = (size_t*)malloc(vec_size * sizeof(size_t));
        for (size_t j = 0; j < vec_size; j++) {
            tmp[j] = encoded[i * vec_size + j];
        }
        *(vectors + i) = (struct vec){ vec_size, tmp };
    }
    // умножает векторы на проверочную матрицу, получая массив индексов ошибки
    // (в обратном порядке записанные 3 двоичных символа)
    for (size_t i = 0; i < 5; i++) {
        *(decipher + i) = nonsquare_matr_mul_vec(&H, vectors + i);
    }
    for (size_t i = 0; i < 5; i++) {
        size_t typo = number(decipher[i].v);
        print_vec(&decipher[i]);
        // если есть ошибка, прибавляет 1 и берёт по модулю 2 (1->0, 0->1)
        if (typo != 0) {
            vectors[i].v[typo - 1] += 1;
            vectors[i].v[typo - 1] %= 2;
        }
    }
    return vectors;
}
```

```c
// 35 двоичных символов превращает в слово длины 4
char* decode(size_t* encoded) {
    struct vec* v = check_parity_bits(encoded);
    // выбираем все не проверочные биты
    size_t multiplied[5][4];
    for (size_t i = 0; i < 5; i++) {
        multiplied[i][0] = v[i].v[2];
        multiplied[i][1] = v[i].v[4];
        multiplied[i][2] = v[i].v[5];
        multiplied[i][3] = v[i].v[6];
    }
    // склеиваем 5 массивов длины 4
    size_t* encoded = (size_t*)malloc(20 * sizeof(size_t));
    for (size_t i = 0; i < 5; i++) {
        for (size_t j = 0; j < 4; j++) {
            encoded[i * 4 + j] = multiplied[i][j];
        }
    }
    for (size_t i = 0; i < 4; i++) {
        for (size_t j = 0; j < 5; j++) {
            encoded[i * 5 + j];
        }
    }
    // разбить на 4 буквы
    char* res = (char*)malloc(4 * sizeof(char));
    for (size_t i = 0; i < 4; i++) {
        size_t* tmp = (size_t*)malloc(5 * sizeof(size_t));
        for (size_t j = 0; j < 5; j++) {
            tmp[j] = encoded[i * 5 + j];
        }
        res[i] = get_letter(tmp);
    }
    return res;
}
```

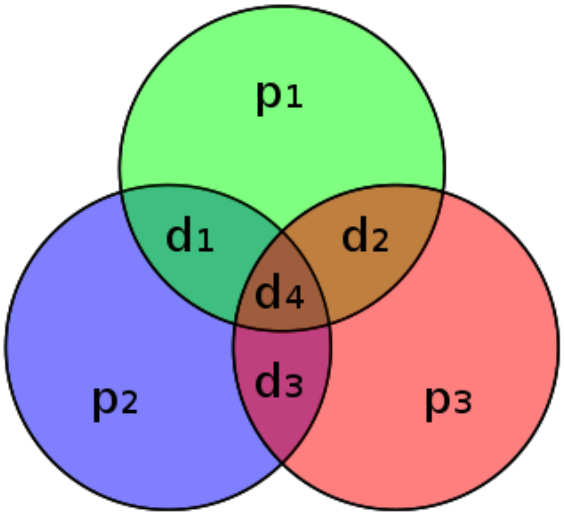
```c
int main() {
    system("chcp 1251");
    srand(clock());
    abc_init();
    char* word = "тома";
    // кодирует слово
    size_t* encoded = encoding(word);
    print_str(encoded, 35);
    printf("\n");
    // добавляет 3 опечатки
    encoded = typo(typo(typo(encoded)));
    print_str(encoded, 35);
    printf("\n");
    // печатает результат декодирования
    print_string(decode(encoded));

    return 0;
}
```

Объяснение, откуда берётся матрица G:

Код Хэмминга состоит из 7 битов, 4 из которых -- биты информации, а остальные 3 -- биты чётности, то есть проверочные биты, позволяющие восстановить, в каком бите была ошибка (только при условии, что в одном блоке из 7ми символов не более одной ошибки). Биты информации начинаются на “d”, биты чётности -- на “p” (см. рис. справа). Почему именно в таком порядке идут биты чётности и биты данных? Можно заметить, что биты чётности имеют номера, являющиеся степень двойки (1, 2, 4). Это стандарт для всех шифров Хэмминга. Благодаря такому расположению каждый бит информации охвачен уникальным наборов битов чётности. Это позволяет однозначно найти ошибочный бит при декодировании. При декодировании биты информации должны оставаться неизменными, поэтому в матрице G их строках ровно одна единица (отмечены зелёным)



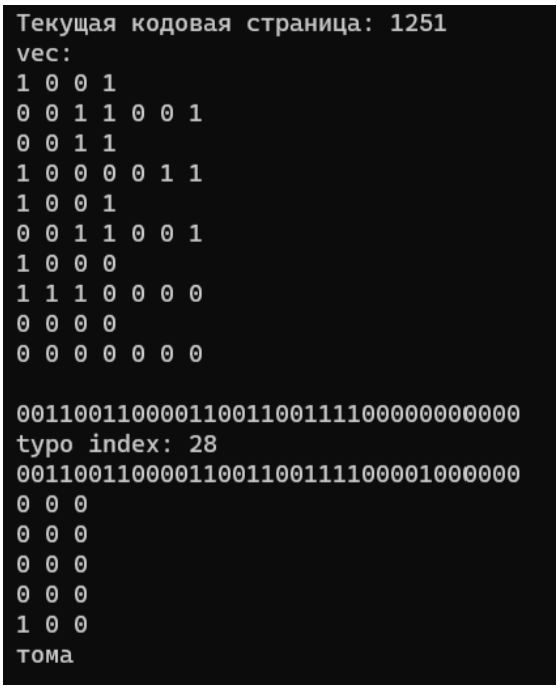
Эта картинка ещё более наглядно показывает, какие биты информации (d) учитывают биты чётности (p).

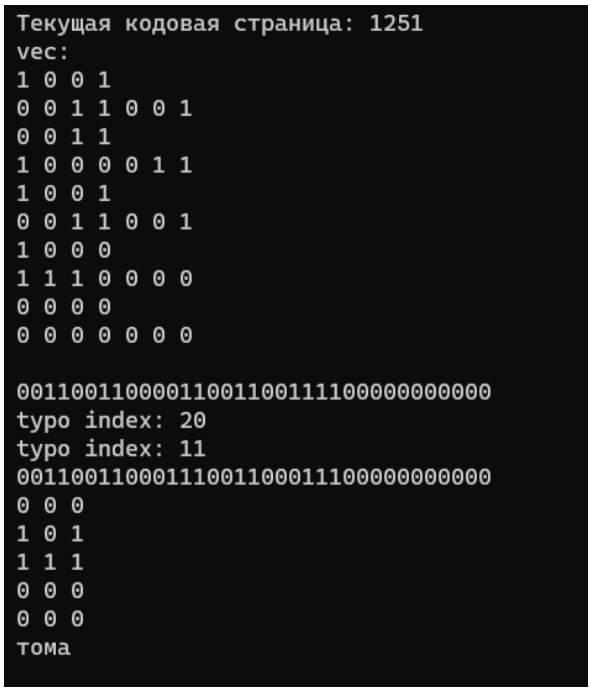


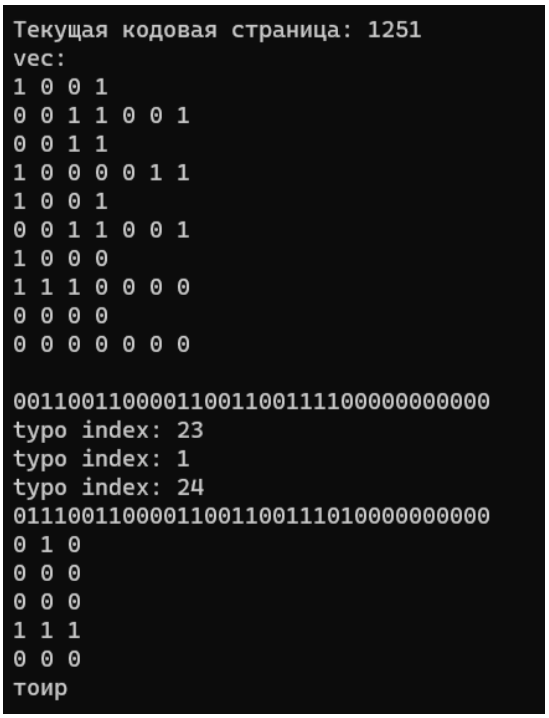
Говоря о матрице H, она просто отображает, за какие биты отвечает какой бит чётности.

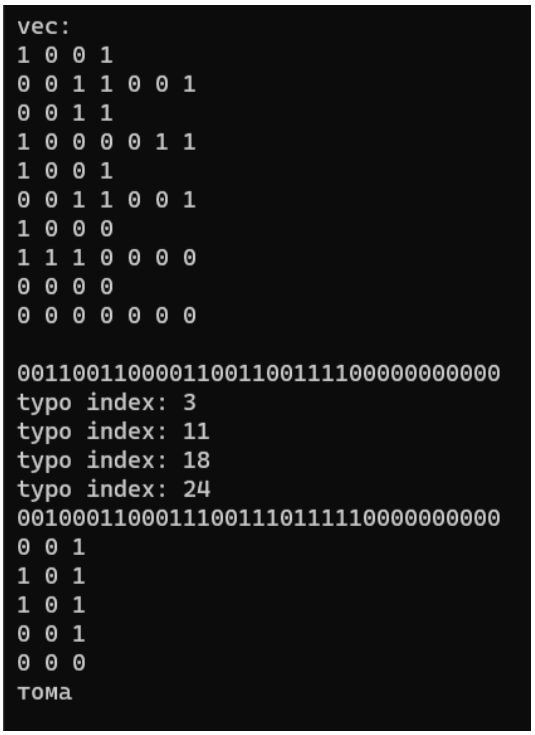
Bit position		1	2	3	4	5	6	7
Encoded data bits		p1	p2	d1	p4	d2	d3	d4
Parity bit	p1	✓		✓		✓		✓
	p2		✓	✓			✓	✓
	p4				✓	✓	✓	✓

Анализ результатов









Здесь представлены результаты для 1, 2, 3 и 4 опечаток соответственно. Сначала выводятся пары исходное-закодированное значение вектора, затем строка из 35 символов, затем добавляются опечатки на указанном индексе и выводится испорченная строка. Потом выводится результат проверки по матрице H. И в конце – исправленное сообщение. У кода Хэмминга есть недостаток: если больше одной опечатки произошло в одном блоке из 7-ми символов кода, найти ошибку мы уже не сможем, что и произошло в предпоследнем случае.

Task 4

Эссе

Задача заключается в следующем. Вам дана доска 8*8, на которой лежат монеты, повёрнутые орлом или решкой. Нужно перевернуть ровно одну монету и тем самым указать на определённую клетку, в которой лежит типо ключ.

Пусть орёл -- 1, решка -- 0. Тогда данную доску с 64 монетами можно представить в виде матрицы 8x8 из нулей и единиц. Теперь можно занумеровать все клеточки. Будьте внимательны, нумерация начинается с нуля, чтобы каждое число было представимо в диапазоне 000000 до 111111 (шестизначные коды).

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Пусть первый заключённый выпишет номера всех клеток, в которых находится единица. Затем нужно сложить по модулю все эти номера в двоичном формате. Напомню таблицу истинности этой операции:

Input		Output
A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Таким образом, мы получим шестизначный двоичный код. И всегда будет существовать единственный код, xor полученного сообщения с которым будет давать номер ячейки. Либо можно будет перевернуть один из орлов таким образом, чтобы убрать из сложения по модулю некоторый код.

Хочу добавить, что так как у нас есть ячейка с кодом 000000, если код суммы всех единиц уже даёт номер нужной ячейки, можно просто перевернуть монету № 0, чтобы ситуация не изменилась.

Также хочу отметить, что эта задача решаема только тогда, когда количество ячеек таблицы является степенью двойки. Иначе не все коды можно будет получить переворачиванием монеты, и некоторые положения ключа просто невозможно будет закодировать.

Also you can find all my code here:

https://github.com/cgsg-tt6ITMO/s3-practlinal-lab1