Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра вычислительных систем

ОТЧЕТ
по практической работе 3

по дисциплине «**Программирование**»

Выполнил:
студент гр. ИС-241
«16» мая 2023 г. _____ /Бондаренко А.А./

Проверил:
ст. преп. кафедры ВС
«__» мая 2023 г. _____ /Фульман В.О./

Оценка «_____»

Новосибирск 2023

ОГЛАВЛЕНИЕ

ЗАДАНИЕ

В первой части вам предлагается разобрать алгоритм кодирования целых чисел, называемый varint (variable integer). Такой способ кодирования позволяет использовать переменное количество байт для представления целых чисел и благодаря этому обеспечивает компактность данных. Вам нужно разработать приложение для записи и чтения чисел в сыром виде и в формате varint, сравнить эти два способа по эффективности и сделать выводы о применимости предложенного способа кодирования.

Во второй части необходимо самостоятельно реализовать алгоритм кодирования UTF-8. Этот алгоритм решает аналогичную задачу — позволяет кодировать целые числа переменным количеством байт, но используется для кодирования кодов символов и поддерживает обратную совместимость с кодировкой ASCII.

Задание 1

Разработайте приложение, которое генерирует 1000000 случайных чисел и записывает их в два бинарных файла. В файл uncompressed.dat запишите числа в несжатом формате, в файл compressed.dat — в формате varint. Сравните размеры файлов.

Реализуйте чтение чисел из двух файлов. Добавьте проверку: последовательности чисел из двух файлов должны совпадать.

Использование формата varint наиболее эффективно в случаях, когда подавляющая доля чисел имеет небольшие значения. Для выполнения работы используйте функцию генерации случайных чисел:

```c
#include <stdint.h>

/*
 * Диапазон              Вероятность
 * --------------------- -----------
 * [0; 128)              90%
 * [128; 16384)          5%
 * [16384; 2097152)      4%
 * [2097152; 268435455) 1%
 */
uint32_t generate_number()
{
    const int r = rand();
    const int p = r % 100;
    if (p < 90) {
        return r % 128;
    }
    if (p < 95) {
        return r % 16384;
    }
    if (p < 99) {
        return r % 2097152;
    }
    return r % 268435455;
}
```

В работе мы будем использовать следующие варианты функций кодирования и декодирования:

```c
#include <assert.h>
#include <stddef.h>
#include <stdint.h>

size_t encode_varint(uint32_t value, uint8_t* buf)
{
    assert(buf != NULL);
    uint8_t* cur = buf;
    while (value >= 0x80) {
        const uint8_t byte = (value & 0x7f) | 0x80;
        *cur = byte;
        value >>= 7;
        ++cur;
    }
    *cur = value;
    ++cur;
    return cur - buf;
}

uint32_t decode_varint(const uint8_t** bufp)
{
    const uint8_t* cur = *bufp;
    uint8_t byte = *cur++;
    uint32_t value = byte & 0x7f;
    size_t shift = 7;
    while (byte >= 0x80) {
        byte = *cur++;
        value += (byte & 0x7f) << shift;
        shift += 7;
    }
    *bufp = cur;
    return value;
}
```

Задание 2

Разработать приложение для кодирования и декодирования чисел по описанному выше алгоритму.

Предлагается следующая структура проекта:

```
.
|-- Makefile
`-- src
    |-- coder.c
    |-- coder.h
    |-- command.c
    |-- command.h
    `-- main.c`
```

ВЫПОЛНЕНИЕ РАБОТЫ

Задание 1

Приложение random генерирует 1000000 случайных чисел, записывает их в незакодированном виде и в закодированном в два разных бинарных файла.

После файлы закрываются и открываются снова, но уже в режиме чтения, а не записи:

```
fclose(uncompressed);
fclose(compressed);

uncompressed = fopen("uncompressed.dat", "rb");
compressed = fopen("compressed.dat", "rb");
```

С помощью перемещения указателя внутри файла функция ftell подсчитывает общий размер обоих файлов, чтобы вычислить коэффициент сжатия:

```
fseek(uncompressed, 0, SEEK_END);
long uncomp_size = ftell(uncompressed);
fseek(compressed, 0, SEEK_END);
long comp_size = ftell(compressed);
```

Затем указатели внутри файлов перемещаются к началу файлов, и, используя функцию fread, каждое число из закодированного файла декодируется и сравнивается с изначальным, не кодированным числом. Так реализуется проверка на то, что кодирование прошло корректно и результаты обратной работы совпадают.

Кодирование, декодирование и генерация случайных чисел реализованы с помощью функций, приведённых в Задании 1 (раздел Задание).

Пример результата работы программы:

```
jumkot@Jumkot:~/prog/Lab/Prog_lab/laba3/1$ ls -1
random.c
jumkot@Jumkot:~/prog/Lab/Prog_lab/laba3/1$ gcc -Wall -o random random.c
jumkot@Jumkot:~/prog/Lab/Prog_lab/laba3/1$ ./random
4000000
1159207
Коэффициент сжатия равен 3
jumkot@Jumkot:~/prog/Lab/Prog_lab/laba3/1$ ls -1
compressed.dat
random
random.c
uncompressed.dat
jumkot@Jumkot:~/prog/Lab/Prog_lab/laba3/1$
```

Задание 2

int main(int argc, char* argv[])

Функция main служит головным “переключателем” в программе: с помощью параметров запуска она вызывает одну из “консольных” функций (argv[1]) и передаёт им имена входных (argv[2]) и выходных (argv[3]) файлов.

В случае неправильного написания команды программа возвращает -1 и завершает выполнение с ненулевым кодом с помощью функции exit(). Проверка соответствия написания производится с помощью функции strcmp():

```
if ((strcmp(argv[1], "encode") == 0)) {
    if (encode_file(argv[2], argv[3]) == -1) {
        printf("Ошибка: невозможно кодировать %s в %s\n", argv[2], argv[3]);
        exit(EXIT_FAILURE);
    }
```

При получении неверного количества аргументов программа выдаёт ошибку по приведённому в лабораторной работе формату и завершает выполнение тем же способом.

```
void print_usage(char* argv[])
{
    printf("Usage:\n");
    printf("%s encode <in-file-name> <out-file-name>\n", argv[0]);
    printf("%s decode <in-file-name> <out-file-name>\n", argv[0]);
    exit(EXIT_FAILURE);
}
```

Примеры результатов работы программы (при приведённых в примере входных данных):

```
jumkot@Jumkot:~/prog/Lab/Prog_lab/laba3/2/bin$ ./coder encode points.txt units.bin
jumkot@Jumkot:~/prog/Lab/Prog_lab/laba3/2/bin$ hexdump -C units.bin
00000000  07 c7 a7 e7 a7 a7 f7 a7  a7 a7                    |..........|
0000000a
jumkot@Jumkot:~/prog/Lab/Prog_lab/laba3/2/bin$
```

```
./coder decode units.bin results.txt
```

```
1   7
2   1e7
3   79e7
4   1e79e7
5
```

results.txt

int encode(uint32_t code_point, CodeUnits* code_unit)

В функции encode сначала производится проверка корректности числа – если переданное макросу assert выражение false, то он выведет ошибку в stderr (стандартный поток вывода ошибок).

Далее следует конструкция из if-ов:

1) если полученное число меньше 10000000 в двоичной системе – 7 значащих бит
   Тогда фактическая длина массива будет 1, а число целиком без преобразований переносится в нулевой по индексу элемент массива.
2) если полученное число меньше 100000 000000 в двоичной системе – 11 значащих бит (т.к. если число кодируется двумя байтами и более, то значащих бит остаётся не более 6 вместо 8 в каждой части)
   В таком случае длина будет уже 2. После в нулевой элемент массива заносится исходное число, битово умноженное на 0x0c (110xxxxx) для обозначения количества последующий байт, с происходит сдвиг на 6 бит вправо. Следующий элемент битово умножается на 0x3f (00111111) для преобразования, и сложение с 0x80 для добавления 10 в начале.
3) если полученное число меньше 10000 000000 000000 в двоичной системе – 16 значащих бит
4) если полученное число меньше 1000 000000 000000 000000 в двоичной системе – 21 значащий бит

В третьем и четвёртом else if длина массива будет 3 и 4 соответственно, сдвиг будет на 12 и на 18 бит соответственно (с уменьшением на 6 на каждом шаге вплоть до последнего), а сложение первого байта будет происходить соответственно с 1110xxxx и 11110xxx.

```
} else if (code_point < 0x200000) {
    code_unit->length = 4;
    code_unit->code[0] = 0xf0 | (code_point >> 18);
    code_unit->code[1] = 0x80 | ((code_point >> 12) & 0x3f);
    code_unit->code[2] = 0x80 | ((code_point >> 6) & 0x3f);
    code_unit->code[3] = 0x80 | (code_point & 0x3f);
```

uint32_t decode(const CodeUnits* code_unit)

В функции decode происходит декодирование исходя из длины массива:

1. Длина 1
   Число переносится без изменений (нет служебных битов).
2. Длина 2
   Первый элемент переносится в массив с умножением на 00011111 (три служебных бита), а каждый последующий байт переносится со сдвигом влево на 6 и умножением на 0x3f (00111111) (потеря служебной 10 в начале).

```
} else if (code_unit->length == 2) {
    code_point = code_unit->code[0] & 0x1f;
    code_point = (code_point << 6) | (code_unit->code[1] & 0x3f);
```

3. Длина 3
4. Длина 4

При длинах 3 и 4 все шаги такие же, только умножение первого байта происходит на 00001111 и на 00000111 соответственно, потому что больше служебных бит.

int write_code_unit(FILE* out, const CodeUnits* code_unit)

Функция производит запись закодированных по предложенному шаблону данных в бинарный файл.

В функции write_code_unit производится проверка, что ни один из указателей не NULL и длина массива с байтами не равна нуля. Если хотя бы одно из условий не выполняется, функция вернёт -1.

Если проверки прошли успешно, то с помощью fwrite в выходной файл записывается содержимое изначального массива в соответствии с его длиной. В случае, если fwrite запишет меньше, чем переданное ей code_unit->length, то функция также вернёт -1.

int read_next_code_unit(FILE* in, CodeUnits* code_unit)

Функция производит чтение закодированных по предложенному шаблону данных из бинарного файла.

Функция read_next_code_unit сначала проверяет, не является ли указатель на файл нулевым. Также проверяется, чтобы fread возвращала указанное при вызове количество объектов (один), и если это не так, достигнут ли конец файла – тогда возвращается -1. Вместе с этим производится занесение кодированного числа из файла в массив.

Далее идёт бесконечный цикл while. Первый if проверяет, является ли первый элемент и последним (т.е. отсутствуют служебные единицы) с помощью умножения на 0x80 (10000000). Если да, то длина массива обозначается 1, и работа функции завершается.

После блоками из else if производится определение длины массива (количества байт в кодированном числе) посредством битового сдвига нулевого элемента и сравнения с соответствующими шестнадцатеричными числами:

1. При двух байтах сдвиг на пять битов и сравнение с 0x6 (0110)
2. При трёх байтах сдвиг на четыре бита и сравнение с 0xe (1110)
3. При четырёх байтах сдвиг на три бита и сравнение с 0x1e (11110)

```
else if (code_unit->code[0] >> 5 == 0x6) {
  if (fread(&(code_unit->code[1]), 1, 1, in) != 1) {
      continue;
  }
  if (code_unit->code[1] >> 6 != 0x2) {
      code_unit->code[0] = code_unit->code[1];
      continue;
  }
  code_unit->length = 2;

  return 0;
```

Если в процессе чтения происходит ошибка (если следующий байт битый и не соответствует формату), функция пропускает его и продолжает чтение следующего байта, пока не будет найден корректный элемент, переходя к следующему шагу цикла с помощью оператора continue. Тогда длина массива устанавливается в соответствии с else if, в который вошла функция, и функция возвращает 0.

Когда программа дойдёт до конца любого из else if, работа функции закончится.

```
int encode_file(const char* in_file_name, const char* out_file_name)
```

“Консольная” функция, которая используется для выбора команды при запуске программы в качестве передаваемого параметра. Отвечает за кодирование текстового файла в бинарный, делая это с помощью последовательного вызова вышеописанных функций.

Пока не будет достигнут конец файла, шестнадцатеричные числа из текстового файла будут по одному считываться и передаваться в функцию encode для кодирования, после – заноситься в результирующий бинарный файл с помощью write_code_unit. В случае, если значение внутри входного файла не соответствует формату uint32_t, программа выдаст ошибку и завершится.

Также до начала непосредственной работы производится проверка на корректное открытие входного и выходного файлов, полученных как аргументы.

```
int decode_file(const char* in_file_name, const char* out_file_name)
```

“Консольная” функция, которая используется для выбора команды при запуске программы в качестве передаваемого параметра. Отвечает за декодирование бинарного файла в бинарный, делая это с помощью последовательного вызова вышеописанных функций.

Пока не будет достигнут конец файла, закодированные по шаблону числа из бинарного файла будут по одному передаваться в функцию decode для декодирования и записываться в выходной текстовый файл с помощью read_next_code_unit.

До начала непосредственной работы производится проверка на корректное открытие входного и выходного файлов, полученных как аргументы.

ПРИЛОЖЕНИЕ

Приложение 1

random.c

```c
#include <assert.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

size_t encode_varint(uint32_t value, uint8_t* buf)
{
    assert(buf != NULL);
    uint8_t* cur = buf;
    while (value >= 0x80) {
        const uint8_t byte = (value & 0x7f) | 0x80;
        *cur = byte;
        value >>= 7;
        ++cur;
    }
    *cur = value;
    ++cur;
    return cur - buf;
}

uint32_t decode_varint(const uint8_t** bufp)
{
    const uint8_t* cur = *bufp;
    uint8_t byte = *cur++;
    uint32_t value = byte & 0x7f;
    size_t shift = 7;
    while (byte >= 0x80) {
        byte = *cur++;
        value += (byte & 0x7f) << shift;
        shift += 7;
    }
    *bufp = cur;
    return value;
}

/*
* Диапазон              Вероятность
* --------------------- -----------
* [0; 128)              90%
* [128; 16384)          5%
* [16384; 2097152)      4%
* [2097152; 268435455) 1%
*/

uint32_t generate_number()
{
    const int r = rand();
    const int p = r % 100;
    if (p < 90) {
        return r % 128;
    }
```

```
    if (p < 95) {
        return r % 16384;
    }
    if (p < 99) {
        return r % 2097152;
    }
    return r % 268435455;
}

int main()
{
    FILE* uncompressed = fopen("uncompressed.dat", "wb");
    FILE* compressed = fopen("compressed.dat", "wb");
    uint32_t number;
    uint32_t varint_size;
    uint8_t* var_arr = malloc(sizeof(uint32_t));

    for (int i = 0; i < 1000000; i++) {
        number = generate_number();
        fwrite(&number, sizeof(number), 1, uncompressed);
        varint_size = encode_varint(number, var_arr);
        fwrite(var_arr, varint_size, 1, compressed);
    }

    fclose(uncompressed);
    fclose(compressed);

    uncompressed = fopen("uncompressed.dat", "rb");
    compressed = fopen("compressed.dat", "rb");

    fseek(uncompressed, 0, SEEK_END);
    long uncomp_size = ftell(uncompressed);
    fseek(compressed, 0, SEEK_END);
    long comp_size = ftell(compressed);

    printf("%ld\n", uncomp_size);
    printf("%ld\n", comp_size);

    printf("Коэффициент сжатия равен %ld\n", (uncomp_size / comp_size));

    fseek(uncompressed, 0, SEEK_SET);
    fseek(compressed, 0, SEEK_SET);

    uint32_t decode;
    var_arr = realloc(var_arr, comp_size);
    const uint8_t* buf = var_arr;
    fread(var_arr, sizeof(uint8_t), comp_size, compressed);

    for (int i = 0; i < 1000000; i++) {
        fread(&number, sizeof(uint32_t), 1, uncompressed);
        decode = decode_varint(&buf);
        if (number != decode) {
            printf("Не удалось произвести кодирование или декодирование (%d)\n",
i);
            return 1;
        }
    }

```

```
111     free(var_arr);
112     fclose(uncompressed);
113     fclose(compressed);
114 
115     return 0;
116 }
```

Приложение 2

main.c

```
 1 #include <stdint.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 
 5 #include "coder.h"
 6 #include "command.h"
 7 
 8 void print_usage(char* argv[]);
 9 
10 int main(int argc, char* argv[])
11 {
12     if (argc != 4)
13         print_usage(argv);
14 
15     if ((strcmp(argv[1], "encode") == 0)) {
16         if (encode_file(argv[2], argv[3]) == -1) {
17             printf("Ошибка: невозможно кодировать %s в %s\n", argv[2], argv[3]);
18             exit(EXIT_FAILURE);
19         }
20     } else if (strcmp(argv[1], "decode") == 0) {
21         if (decode_file(argv[2], argv[3]) == -1) {
22             printf("Ошибка: невозможно декодировать %s в %s\n", argv[2], argv[3]);
23             exit(EXIT_FAILURE);
24         }
25     } else {
26         print_usage(argv);
27     }
28 
29     return 0;
30 }
31 
32 void print_usage(char* argv[])
33 {
34     printf("Usage:\n");
35     printf("%s encode <in-file-name> <out-file-name>\n", argv[0]);
36     printf("%s decode <in-file-name> <out-file-name>\n", argv[0]);
37     exit(EXIT_FAILURE);
38 }
```

coder.c

```
 1 #include <assert.h>
 2 
 3 #include "coder.h"
```

```c
#include "command.h"

int encode(uint32_t code_point, CodeUnits* code_unit)
{
    assert(code_unit != NULL);

    if (code_point < 0x80) {
        code_unit->length = 1;
        code_unit->code[0] = code_point;
    } else if (code_point < 0x800) {
        code_unit->length = 2;
        code_unit->code[0] = 0xc0 | (code_point >> 6);
        code_unit->code[1] = 0x80 | (code_point & 0x3f);
    } else if (code_point < 0x10000) {
        code_unit->length = 3;
        code_unit->code[0] = 0xe0 | (code_point >> 12);
        code_unit->code[1] = 0x80 | ((code_point >> 6) & 0x3f);
        code_unit->code[2] = 0x80 | (code_point & 0x3f);
    } else if (code_point < 0x200000) {
        code_unit->length = 4;
        code_unit->code[0] = 0xf0 | (code_point >> 18);
        code_unit->code[1] = 0x80 | ((code_point >> 12) & 0x3f);
        code_unit->code[2] = 0x80 | ((code_point >> 6) & 0x3f);
        code_unit->code[3] = 0x80 | (code_point & 0x3f);
    } else {
        printf("Ошибка: %d превышает максимальное значение для кодирования\n",
code_point);
        code_unit->length = 0;

        return -1;
    }
    return 0;
}

uint32_t decode(const CodeUnits* code_unit)
{
    uint32_t code_point;

    if (code_unit->length == 1) {
        code_point = code_unit->code[0];
    } else if (code_unit->length == 2) {
        code_point = code_unit->code[0] & 0x1f;
        code_point = (code_point << 6) | (code_unit->code[1] & 0x3f);
    } else if (code_unit->length == 3) {
        code_point = code_unit->code[0] & 0xf;
        code_point = (code_point << 6) | (code_unit->code[1] & 0x3f);
        code_point = (code_point << 6) | (code_unit->code[2] & 0x3f);
    } else if (code_unit->length == 4) {
        code_point = code_unit->code[0] & 0x7;
        code_point = (code_point << 6) | (code_unit->code[1] & 0x3f);
        code_point = (code_point << 6) | (code_unit->code[2] & 0x3f);
        code_point = (code_point << 6) | (code_unit->code[3] & 0x3f);
        code_unit = NULL;
    }

    return code_point;
}

```

```c
int write_code_unit(FILE* out, const CodeUnits* code_unit)
{
    if ((out == NULL) || (code_unit == NULL) || (code_unit->length == 0)) {
        return -1;
    }

    if (fwrite(code_unit->code, 1, code_unit->length, out) < code_unit->length) {
        return -1;
    }

    return 0;
}

int read_next_code_unit(FILE* in, CodeUnits* code_unit)
{
    if (in == NULL) {
        return -1;
    }

    if (fread(code_unit->code, 1, 1, in) != 1) {
        if (feof(in)) {
            return -1;
        }
    }

    while (1) {
        if (feof(in)) {
            code_unit = NULL;
            return -1;
        }
        if ((code_unit->code[0] & 0x80) == 0) {
            code_unit->length = 1;
            return 0;
        } else if (code_unit->code[0] >> 5 == 0x6) {
            if (fread(&(code_unit->code[1]), 1, 1, in) != 1) {
                continue;
            }
            if (code_unit->code[1] >> 6 != 0x2) {
                code_unit->code[0] = code_unit->code[1];
                continue;
            }
            code_unit->length = 2;

            return 0;
        } else if (code_unit->code[0] >> 4 == 0xe) {
            if (fread(&(code_unit->code[1]), 1, 1, in) != 1) {
                continue;
            }
            if (code_unit->code[1] >> 6 != 0x2) {
                code_unit->code[0] = code_unit->code[1];
                continue;
            }
            if (fread(&(code_unit->code[2]), 1, 1, in) != 1) {
                continue;
            }
            if (code_unit->code[2] >> 6 != 0x2) {
                code_unit->code[0] = code_unit->code[2];
                continue;
```

```
            }
            code_unit->length = 3;
            return 0;
        } else if (code_unit->code[0] >> 3 == 0x1e) {
            if (fread(&(code_unit->code[1]), 1, 1, in) != 1) {
                continue;
            }
            if (code_unit->code[1] >> 6 != 0x2) {
                code_unit->code[0] = code_unit->code[1];
                continue;
            }
            if (fread(&(code_unit->code[2]), 1, 1, in) != 1) {
                continue;
            }
            if (code_unit->code[2] >> 6 != 0x2) {
                code_unit->code[0] = code_unit->code[2];
                continue;
            }
            if (fread(&(code_unit->code[3]), 1, 1, in) != 1) {
                continue;
            }
            if (code_unit->code[3] >> 6 != 0x2) {
                code_unit->code[0] = code_unit->code[3];
                continue;
            }
            code_unit->length = 4;

            return 0;
        }
        fread(code_unit->code, 1, 1, in);
    }
}
```

coder.h

```
#pragma once

#include <stdint.h>
#include <stdio.h>

enum {
    MaxCodeLength = 4
};

typedef struct {
    uint8_t code[MaxCodeLength];
    size_t length;
} CodeUnits;

int encode(uint32_t code_point, CodeUnits* code_units);
uint32_t decode(const CodeUnits* code_unit);
int write_code_unit(FILE* out, const CodeUnits* code_unit);
int read_next_code_unit(FILE* in, CodeUnits* code_units);
```

command.c

```c
#include "coder.h"
#include "command.h"

int encode_file(const char* in_file_name, const char* out_file_name)
{
    FILE *in, *out;
    in = fopen(in_file_name, "r");
    if (!in) {
        printf("Ошибка: не удалось открыть файл %s\n", in_file_name);
        return -1;
    }

    out = fopen(out_file_name, "wb");
    if (!out) {
        printf("Ошибка: не удалось открыть файл %s\n", out_file_name);
        fclose(in);
        return -1;
    }

    uint32_t value;
    CodeUnits enc_units;

    while (!feof(in)) {
        if (fscanf(in, "%" SCNx32, &value) == 0) {
            printf("Ошибка: элемент '%d'в файле '%s' не является значением типа
uint32_t\n", value, in_file_name);
            return -1;
        }
        encode(value, &enc_units);
        write_code_unit(out, &enc_units);
    }

    fclose(in);
    fclose(out);

    return 0;
}

int decode_file(const char* in_file_name, const char* out_file_name)
{
    FILE *in, *out;
    in = fopen(in_file_name, "rb");
    if (!in) {
        printf("Ошибка: не удалось открыть файл %s\n", in_file_name);
        return -1;
    }

    out = fopen(out_file_name, "w");
    if (!out) {
        printf("Ошибка: не удалось открыть файл %s\n", out_file_name);
        fclose(in);
        return -1;
    }

    uint32_t value;
    CodeUnits dec_units;
```

```
57
58      while (!feof(in)) {
59          if (read_next_code_unit(in, &dec_units) == EOF) {
60              break;
61          }
62          value = decode(&dec_units);
63          fprintf(out, "%" PRIx32, value);
64          fprintf(out, "\n");
65      }
66
67      fclose(in);
68      fclose(out);
69
70      return 0;
71  }
```

command.h

```
1  #pragma once
2
3  #include <inttypes.h>
4
5  #include "coder.h"
6
7  int encode_file(const char* in_file_name, const char* out_file_name);
8  int decode_file(const char* in_file_name, const char* out_file_name);
```