Лабораторна робота №1

Завдання до виконання у лабораторноій роботі

1. Декодування та аналіз строки UTF-8, написаній на укр. мові (в >= 120 симовлів).
2. Алгоритм компресії даних Shannon(a)-Fano, кодувати та декодувати строку з 1 завдання
3. Алгоритм компресії даних Huffman(a), кодувати та декодувати строку з 1 завдання

Вимоги до завданнь

- вивести показники ефективності стиснення (кодування) для кожного формату кодування використовуючи формули аналізу Крафта та Шеннона
- Для кожного завдання вивести кількість байт, та ширину бітового рядка (bitwidth)
- Для 1 завдання вивести кількість слів, символів UTF-8, провести аналіз частоти появи кожного символу у гістограммі (графіку), побудувати код Грея (Gray`s code)
- Для 2 та 3 завдання вивести кофіцієнт стиснення без урахування символьного дерева (тобто не враховуючи кількості байт дерева)

Завдання 1

Кроки роботи программи:

1. Прочитати строку з файлу.
2. Вивести строку на екран
3. Вивести байти строки в їх реальному вигляді (закодованому)
4. Вивести байти як кодові числа (HEX числа для символів в натуральному вигляді)
5. Вивести бінарний вигляд кожного байту
6. Вивести код грея кожного байту
7. Вивести кожен символ строки
8. Відсортувати та вивести символи за їх частотою у строці
9. Вивести справку з усією числовою інформацією про строку
10. Звільнити пам’ять

Увесь проект цілком можна будет перегланути в доданому архіві 'sourcecode.zip'

Код программи на C:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <math.h>

#define MAX_STRING_SIZE 256
```

```
#define EOL '\n'
#define END NULL
#define PRINT_AS_CODEPOINTS 0
#define PRINT_AS_BYTES      1

#define HEAD_4BYTE_BITMASK 0x07
#define HEAD_3BYTE_BITMASK 0x0F
#define HEAD_2BYTE_BITMASK 0x1F

#define PAYLOAD_BITMASK 0x3F

#define U8_GRAPHEME_SIZE 4

// macro for debugging
// #define DEBUG

// generic termaintion function
void die(char* error) {
  fprintf(stderr,"%s", error);
  exit(-1);
}

// generic swap for numbers
void swap(int *p1, int *p2) {
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}


// loads the string from a file,
// expects the string to be within 256 bytes.
//
// On error terminates with -1
//
// On string bigger than a buffer, prints warning
char* load_string(char* fpath) {
  FILE* f = fopen(fpath, "r");
  char* string_buffer = malloc(MAX_STRING_SIZE);
  memset(string_buffer, 0, MAX_STRING_SIZE);

  if (!f) {
    free(string_buffer);
    die("failed to open file!");
  }

  if (!string_buffer) {
    free(string_buffer);
    die("failed to use malloc! buy more ram!?");
  }
```

```
  for (uint i = 0; i < MAX_STRING_SIZE; i++)
  {
    char c = fgetc(f);

    if (c == EOF || c == EOL)
      break;

    sprintf(string_buffer,"%s%c",string_buffer, c);
  }

  return string_buffer;
}

// gets length of the string taking UTF-8 in
// consideration.
//
// does not check for encoding validity
int u8strlen(const char *s)
{
  int len=0;
  while (*s) {
    if ((*s & 0xC0) != 0x80)
      len++;
    s++;
  }
  return len;
}

// Get number of bytes in UTF-8 grapheme
uint u8_nbyte(const char *u8char) {
    const uint8_t *byte = (const uint8_t *)u8char;

  // UTF-8 encoding:
  //
  // 1   byte: Head
  // 0-3 byte: Payload
  //
  // IN case of grapheme taking only one byte,
  // treat it like int_8:
  //  0xxxxxxx
  //  ^~~~~~~--> Payload
  //  |
  //  |
  //  means unsighned
  //
  // encoding table:
  //
  //  BYTE1    BYTE2    BYTE3    BYTE4
  //
  // [0xxxxxxx]
  // [110xxxxx 10xxxxxx]
```

```
  // [1110xxxx 10xxxxxx 10xxxxxx]
  // [11110xxx 10xxxxxx 10xxxxxx 10xxxxxx]
  //
  // UTF-8

  int num_bytes = 0;
    if ((*byte & 0x80) == 0x00) { // checking for pattern:        10xxxxxx
        num_bytes = 1;
    } else if ((*byte & 0xE0) == 0xC0) { // checking for pattern: 110xxxxx
        num_bytes = 2;
    } else if ((*byte & 0xF0) == 0xE0) { // checking for pattern: 1110xxxx
        num_bytes = 3;
    } else if ((*byte & 0xF8) == 0xF0) { // checking for pattern: 1111xxxx
        num_bytes = 4;
    }

    return num_bytes;
}

// Decode UTF-8 grapheme -> src string, with known amount of bytes -> nbyte
int u8_char_decode(const char *src ,uint nbyte) {

  const uint8_t *byte = (const uint8_t *)src;
  uint codepoint = 0;

    // Decode the UTF-8 sequence the rough way
    switch (nbyte) {
        case 1:
            codepoint = *byte;
            break;
        case 2:
            codepoint = (*byte & HEAD_2BYTE_BITMASK) << 6;
            byte++;
            codepoint |= (*byte & PAYLOAD_BITMASK);
            break;
        case 3:
            codepoint = (*byte & HEAD_3BYTE_BITMASK) << 12;
            byte++;
            codepoint |= (*byte & PAYLOAD_BITMASK) << 6;
            byte++;
            codepoint |= (*byte & PAYLOAD_BITMASK);
            break;
        case 4:
            codepoint = (*byte & HEAD_4BYTE_BITMASK) << 18;
            byte++;
            codepoint |= (*byte & PAYLOAD_BITMASK) << 12;
            byte++;
            codepoint |= (*byte & PAYLOAD_BITMASK) << 6;
            byte++;
            codepoint |= (*byte & PAYLOAD_BITMASK);
            break;
```

```
        default:
            printf("Invalid UTF-8 sequence\n");
            break;
    }
  return codepoint;
}

// Get codepoint for UTF-8 grapheme using 2 oftenly used functions
uint u8char(const char *u8char)
{
    uint codepoint = 0;
    uint num_bytes = u8_nbyte(u8char);
    return u8_char_decode(u8char, num_bytes);
}




// convert to codepoint array all characters in UTF-8 string
uint* u8_as_codepoint_array(const char *u8_str) {

  uint* buffer = malloc( (u8strlen(u8_str) + 1)  * sizeof(uint));
  uint codepoint;

  buffer[u8strlen(u8_str)] = 0;

  uint u8len = u8strlen(u8_str);

#ifdef DEBUG
  printf("LEN OF U8STR is = %d\t ",u8len);
#endif

  for(uint i = 0; i < u8len; i++)
  {
    codepoint = u8char(u8_str);
    uint symsize = u8_nbyte(u8_str);

#ifdef DEBUG
    printf("codepoint: `%d` with size: %d\n",codepoint, symsize);
#endif

    buffer[i] = codepoint;
    for(uint j = 0; j < symsize; j++)
      u8_str++;
  }
  return buffer;
}


// convert to array of char* , all characters in UTF-8 string
char** u8_as_str_array(const char *u8_str) {
```

```
  char** buffer = malloc( ( u8strlen(u8_str) + 1 ) * sizeof(char*));
   buffer[u8strlen(u8_str)] = END; // easier to iterate over like a string.

  for (uint i = 0; i < u8strlen(u8_str); i++) {
    buffer[i] = malloc( sizeof(char) * U8_GRAPHEME_SIZE); // its 4 bytes.
    memset(buffer[i], 0, 4);
  }
  uint buffer_i = 0;
  uint index = 0;
  uint codepoint;
  uint bytecount;


  uint u8len = u8strlen(u8_str);

#ifdef DEBUG
  printf("LEN OF U8STR is = %d\t ",u8len);
#endif

  for(uint i = 0; i < u8len; i++)
  {
    codepoint = u8char(u8_str);
    uint symsize = u8_nbyte(u8_str);

#ifdef DEBUG
    printf("char : `%d` with size: %d\n",codepoint, symsize);
#endif

    for(uint j = 0; j < symsize; j++)  {
      sprintf(buffer[i],"%s%c", buffer[i],*u8_str);
      u8_str++;
    }
  }
  return buffer;
}


uint u8_words(const char* str){
  uint wc = 0;
  uint seen_letter = 0;

  for (uint i = 0; str[i] != 0; i++)
  {
    if (str[i] != ' ')
      seen_letter = 1;


    if (str[i] == ' '
        || str[i] == '\n'
        || str[i] == '\t'
```

```
        || str[i] == '\0'
        && seen_letter
       )
    {
      wc++;
    }
  }

  if (seen_letter && wc >= 1)
    wc ++;
  else if (seen_letter)
    wc ++;

  return wc;
}


// sort
void sort(uint* arr, int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

float self_information(float chance) {
  return ( -log2(chance) );
}


float shannons_value(float entropy, uint wordcount)
{
  return entropy / wordcount;
}

double krafts_value(uint symbols_count, uint alphabet_size)
{
  return pow(2, -symbols_count) * alphabet_size;
}


// calculate max occurances of an item
// create and array of apropriate size
```

```
// put each occurances to its apropriate location
// in size array.
uint* frequency_array(uint* codepoints, uint* unique_codepoint, float* entropy) {

  uint len = 0;
  uint last_codepoint = 0;
  uint cur_count = 0;
  uint max_count = 0;

  for(uint i = 0; codepoints[i] != 0; i++)
    len++;

  sort(codepoints, len);

  // find max_count of codepoint
  for(uint i = 0; codepoints[i] != 0; i++) {
    uint el = codepoints[i];

    if (cur_count > max_count)
      max_count = cur_count;

    if (el == last_codepoint)
      cur_count++;
    else {
      last_codepoint = el;
      cur_count = 1;
    }
  }

  *unique_codepoint = max_count;

  // create buffer and memeset it to 0
  last_codepoint = 0;
  // cur_count = 1;
  const uint EMPTY = 1;

  uint* count_array = malloc(sizeof(uint) * (max_count+1+1));
  for(uint i = 0; i < max_count; i++)
    count_array[i] = 1;
  count_array[max_count+1] = 0;

  // assign each occurance to its count (index)
  for(uint i = 0; codepoints[i] != 0; i++) {
    uint el = codepoints[i];

    if (el == last_codepoint)
      cur_count++;
    else {
      if (last_codepoint != 0) {
        printf("\n\tcodepoint (%d) was found -> %d",last_codepoint,cur_count);
        float chance = (float)cur_count / (float)len;
```

```
        (*entropy) +=  chance * self_information(chance);
      }
      count_array[cur_count] = last_codepoint;
      last_codepoint = el;
      cur_count = 1;
    }
  }

  return count_array;
}



// print array of strings (each one is UTF-8 character)
void array_print(char** arr) {
  printf("\t");
    for(uint i = 0; arr[i] != 0; i++)
      if (strcmp(arr[i], " ") == 0)
        printf("\n\t");
      else
        printf("[%s] ",arr[i]);
}

// print values as codepoint or grouped decimal binary values
void u8_print(const char *u8_str, int flags) {
  uint codepoint;
  uint bytecount;

  char* type = (flags == PRINT_AS_CODEPOINTS)? "codepoint" : "raw bytes";

  printf(
          "\n\n\n"
          "\t\t UTF-8 string encoded as decimal - "
          "%s:\n\n", type
  );

  uint u8len = u8strlen(u8_str);


  for(uint i = 0; i < u8len; i++)
  {
    codepoint = u8char(u8_str);
    uint symsize = u8_nbyte(u8_str);
    if (flags == PRINT_AS_CODEPOINTS)
        printf("\t\t[%d]",codepoint);
    else if (flags == PRINT_AS_BYTES)
        printf("\t\t[");
    for(uint j = 0; j < symsize; j++) {
        if (flags == PRINT_AS_BYTES)
            printf("[%d]",*u8_str);
```

```
        u8_str++;
    }

    if (flags == PRINT_AS_BYTES)
        printf("] -> \t char[%d]\n",i);
  }

  printf("\n\n\n");

}



// convert binary uint to number
uint binary_to_decimal(uint binary) {
  uint decimal = 0;
  uint weight  = 1;
  uint rem     = 0;

  while(binary != 0)
  {
    rem      =  binary % 10;
    decimal  += rem * weight;
    binary   /= 10;
    weight   *= 2;
  }

  return decimal;
}

// convert decimal to uint binary
void binary(uint8_t decimal) {
   printf(" ");
  int numBits = sizeof(decimal) * 8;

  // Loop through each bit from left to right
  for (int i = numBits - 1; i >= 0; i--) {
    // Use bitwise AND to check if the bit is 1 or 0
    if (decimal & (1 << i)) {
      printf("1");
    } else {
      printf("0");
    }
  }
  printf(" ");
}

// print grays of a decimal
void grays(uint8_t decimal)
{
  uint8_t grays = decimal ^ (decimal >> 1);
```

```
  binary(grays);
}


// print decimal content of string
void slice_print(char* string)
{
    printf("\n\n\t\t "
            "Raw decimal values of incoded UTF-8 string"
            "(negative indicate non ASCII characters): \n\n"
    );

    for(uint i = 0; i < strlen(string); i++)
        printf("[%d]\t",(uint)string[i]);
    printf("\n");
}

// print each codepoint as binary
void binary_print(char* string)
{
  printf("\n\t\tBinary of string:\n");

  const uint8_t *byte = (const uint8_t *)string;
  for(uint i = 0; byte[i] != '\0'; i++)
  {
    if (i % 10 == 0)
      printf("\n");
    binary(byte[i]);
  }
  printf("\n\n");
}


// print grays content of a string
void grays_print(char* string) {

  printf("\n\t\tGrays Binary of string:\n");

  const uint8_t *byte = (const uint8_t *)string;
  for(uint i = 0; byte[i] != '\0'; i++)
  {
    if (i % 10 == 0)
      printf("\n");
    grays(byte[i]);
  }
  printf("\n\n");
}

int main(int argc, char** argv) {

  if (argc != 2)
```

```
  die("Not enough arguments,"
      "expected file path to "
      "be provided as first "
      "argument.");

// load str from file
char* string = load_string(argv[1]);


// print str
printf(
    "\n\n"
    "\tString: '%s'"
    "\n\n", string
);

// print utf-8 bytes untoched
slice_print(string);

// print codepoints
u8_print(string, PRINT_AS_CODEPOINTS);

// utf8 string as bytes and Grays code
binary_print(string);
grays_print(string);

uint*  cpa   = u8_as_codepoint_array(string);
char** u8arr = u8_as_str_array(string);

// print utf8 symbols
printf(
    "\n\n"
    "\tString broken down to its symbols:"
    "\n\n"
);
array_print(u8arr);

printf("\n\n\tCODEPOINTS SORTED:\n");
printf("strlen: %d\n", u8strlen(string));

for(uint i = 0; cpa[i] != 0; i++)
  printf("\t%d", cpa[i]);


float entropy = 0.0;
uint unique = 0;
uint* freq = frequency_array(cpa, &unique, &entropy);

printf(
    "\n\n\t" "Chart of codepoints rarity:" "\n"
);
```

```
for(uint i = 0; freq[i]!=0; i++)
  if (freq[i]!=1) {
    unique++;
    printf("\tcodepoint: [ %d ], count ->\t%d\t [",freq[i],i);
    for(uint c =0; c < i; c++)
      printf("#");
    printf("]\n");
  }

const uint UA_UTF8_SIZE     = 11;
const uint UA_ALPHABET_SIZE = 32 + 6;

// Stats
printf(
    "\n\n\n\n"
    "+\t\t" "String information:" "\n"
    "|\t"   "\n"
    "|\t"   "Number of UTF-8 characters  : %d"       "\n"
    "|\t"   "Byte`s taken to store data  : %ld"      "\n"
    "|\t"   "Bit length (bytes * 8)      : %ld"      "\n"
    "|\t"   "Unique characters count     : %d"       "\n"
    "|\t"   "Words in string             : %d"       "\n"
    "|\t"   "Etropy for the string       : %.2f"     "\n"
    "|\t"   "Kraft's value (UA+SYM = 38) : %.13lf"   "\n"
    "|\t"   "Shannons value for UA_UTF8  : %.2f"     "\n"
    "+\n"
    ,

    u8strlen(string) ,
    strlen(string),
    strlen(string)*8,
    unique,
    u8_words(string),
    entropy,
    krafts_value(UA_UTF8_SIZE,UA_ALPHABET_SIZE),
    shannons_value(entropy, UA_UTF8_SIZE)
);


// defer block
if (string)
  free(string);

if (cpa)
  free(cpa);

if (freq)
  free(freq);

if (u8arr) {
```

```
    for(uint i = 0; u8arr[i] != 0; i++)
      free(u8arr[i]);
    free(u8arr);
  }
  //
}
```

Результат роботи (збережений у файл використовуючи pipe operator (>))

```
clang task1.c -o task1 -g -fsanitize=address && ./task1 file.txt


    String: 'Ми любимо їсти кашу! Так казали козаки на русі, для них той смак був
найкращим, він нагадував їх батьківщину, рідну мати, родину.'



         Raw decimal values of incoded UTF-8 string(negative indicate non ASCII
characters):

[-48]   [-100]  [-48]   [-72]   [32]    [-48]   [-69]   [-47]   [-114]  [-48]   [-79]
[-48]   [-72]   [-48]   [-68]   [-48]   [-66]   [32]    [-47]   [-105]  [-47]   [-127]
[-47]   [-126]  [-48]   [-72]   [32]    [-48]   [-70]   [-48]   [-80]   [-47]   [-120]
[-47]   [-125]  [33]    [32]    [-48]   [-94]   [-48]   [-80]   [-48]   [-70]   [32]
[-48]   [-70]   [-48]   [-80]   [-48]   [-73]   [-48]   [-80]   [-48]   [-69]   [-48]
[-72]   [32]    [-48]   [-70]   [-48]   [-66]   [-48]   [-73]   [-48]   [-80]   [-48]
[-70]   [-48]   [-72]   [32]    [-48]   [-67]   [-48]   [-80]   [32]    [-47]   [-128]
[-47]   [-125]  [-47]   [-127]  [-47]   [-106]  [44]    [32]    [-48]   [-76]   [-48]
[-69]   [-47]   [-113]  [32]    [-48]   [-67]   [-48]   [-72]   [-47]   [-123]  [32]
[-47]   [-126]  [-48]   [-66]   [-48]   [-71]   [32]    [-47]   [-127]  [-48]   [-68]
[-48]   [-80]   [-48]   [-70]   [32]    [-48]   [-79]   [-47]   [-125]  [-48]   [-78]
[32]    [-48]   [-67]   [-48]   [-80]   [-48]   [-71]   [-48]   [-70]   [-47]   [-128]
[-48]   [-80]   [-47]   [-119]  [-48]   [-72]   [-48]   [-68]   [44]    [32]    [-48]
[-78]   [-47]   [-106]  [-48]   [-67]   [32]    [-48]   [-67]   [-48]   [-80]   [-48]
[-77]   [-48]   [-80]   [-48]   [-76]   [-47]   [-125]  [-48]   [-78]   [-48]   [-80]
[-48]   [-78]   [32]    [-47]   [-105]  [-47]   [-123]  [32]    [-48]   [-79]   [-48]
[-80]   [-47]   [-126]  [-47]   [-116]  [-48]   [-70]   [-47]   [-106]  [-48]   [-78]
[-47]   [-119]  [-48]   [-72]   [-48]   [-67]   [-47]   [-125]  [44]    [32]    [-47]
[-128]  [-47]   [-106]  [-48]   [-76]   [-48]   [-67]   [-47]   [-125]  [32]    [-48]
[-68]   [-48]   [-80]   [-47]   [-126]  [-48]   [-72]   [44]    [32]    [-47]   [-128]
[-48]   [-66]   [-48]   [-76]   [-48]   [-72]   [-48]   [-67]   [-47]   [-125]  [46]



         UTF-8 string encoded as decimal - codepoint:

        [1052]      [1080]      [32]        [1083]      [1102]      [1073]      [1080]
[1084]      [1086]      [32]        [1111]      [1089]      [1090]      [1080]
[32]        [1082]      [1072]      [1096]      [1091]      [33]        [32]
[1058]      [1072]      [1082]      [32]        [1082]      [1072]      [1079]
```

```
[1072]      [1083]      [1080]      [32]        [1082]      [1086]      [1079]
[1072]      [1082]      [1080]      [32]        [1085]      [1072]      [32]
[1088]      [1091]      [1089]      [1110]      [44]        [32]        [1076]
[1083]      [1103]      [32]        [1085]      [1080]      [1093]      [32]
[1090]      [1086]      [1081]      [32]        [1089]      [1084]      [1072]
[1082]      [32]        [1073]      [1091]      [1074]      [32]        [1085]
[1072]      [1081]      [1082]      [1088]      [1072]      [1097]      [1080]
[1084]      [44]        [32]        [1074]      [1110]      [1085]      [32]
[1085]      [1072]      [1075]      [1072]      [1076]      [1091]      [1074]
[1072]      [1074]      [32]        [1111]      [1093]      [32]        [1073]
[1072]      [1090]      [1100]      [1082]      [1110]      [1074]      [1097]
[1080]      [1085]      [1091]      [44]        [32]        [1088]      [1110]
[1076]      [1085]      [1091]      [32]        [1084]      [1072]      [1090]
[1080]      [44]        [32]        [1088]      [1086]      [1076]      [1080]
[1085]      [1091]      [46]



        Binary of string:

 11010000  10011100  11010000  10111000  00100000  11010000  10111011  11010001
10001110  11010000
 10110001  11010000  10111000  11010000  10111100  11010000  10111110  00100000
11010001  10010111
 11010001  10000001  11010001  10000010  11010000  10111000  00100000  11010000
10111010  11010000
 10110000  11010001  10001000  11010001  10000011  00100001  00100000  11010000
10100010  11010000
 10110000  11010000  10111010  00100000  11010000  10111010  11010000  10110000
11010000  10110111
 11010000  10110000  11010000  10111011  11010000  10111000  00100000  11010000
10111010  11010000
 10111110  11010000  10110111  11010000  10110000  11010000  10111010  11010000
10111000  00100000
 11010000  10111101  11010000  10110000  00100000  11010001  10000000  11010001
10000011  11010001
 10000001  11010001  10010110  00101100  00100000  11010000  10110100  11010000
10111011  11010001
 10001111  00100000  11010000  10111101  11010000  10111000  11010001  10000101
00100000  11010001
 10000010  11010000  10111110  11010000  10111001  00100000  11010001  10000001
11010000  10111100
 11010000  10110000  11010000  10111010  00100000  11010000  10110001  11010001
10000011  11010000
 10110010  00100000  11010000  10111101  11010000  10110000  11010000  10111001
11010000  10111010
 11010001  10000000  11010000  10110000  11010001  10001001  11010000  10111000
11010000  10111100
 00101100  00100000  11010000  10110010  11010001  10010110  11010000  10111101
00100000  11010000
 10111101  11010000  10110000  11010000  10110011  11010000  10110000  11010000
```

```
10110100  11010001
 10000011  11010000  10110010  11010000  10110000  11010000  10110010  00100000
11010001  10010111
 11010001  10000101  00100000  11010000  10110001  11010000  10110000  11010001
10000010  11010001
 10001100  11010000  10111010  11010001  10010110  11010000  10110010  11010001
10001001  11010000
 10111000  11010000  10111101  11010001  10000011  00101100  00100000  11010001
10000000  11010001
 10010110  11010000  10110100  11010000  10111101  11010001  10000011  00100000
11010000  10111100
 11010000  10110000  11010001  10000010  11010000  10111000  00101100  00100000
11010001  10000000
 11010000  10111110  11010000  10110100  11010000  10111000  11010000  10111101
11010001  10000011
 00101110


        Grays Binary of string:

 10111000  11010010  10111000  11100100  00110000  10111000  11100110  10111001
11001001  10111000
 11101001  10111000  11100100  10111000  11100010  10111000  11100001  00110000
10111001  11011100
 10111001  11000001  10111001  11000011  10111000  11100100  00110000  10111000
11100111  10111000
 11101000  10111001  11001100  10111001  11000010  00110001  00110000  10111000
11110011  10111000
 11101000  10111000  11100111  00110000  10111000  11100111  10111000  11101000
10111000  11101100
 10111000  11101000  10111000  11100110  10111000  11100100  00110000  10111000
11100111  10111000
 11100001  10111000  11101100  10111000  11101000  10111000  11100111  10111000
11100100  00110000
 10111000  11100011  10111000  11101000  00110000  10111001  11000000  10111001
11000010  10111001
 11000001  10111001  11011101  00111010  00110000  10111000  11101110  10111000
11100110  10111001
 11001000  00110000  10111000  11100011  10111000  11100100  10111001  11000111
00110000  10111001
 11000011  10111000  11100001  10111000  11100101  00110000  10111001  11000001
10111000  11100010
 10111000  11101000  10111000  11100111  00110000  10111000  11101001  10111001
11000010  10111000
 11101011  00110000  10111000  11100011  10111000  11101000  10111000  11100101
10111000  11100111
 10111001  11000000  10111000  11101000  10111001  11001101  10111000  11100100
10111000  11100010
 00111010  00110000  10111000  11101011  10111001  11011101  10111000  11100011
00110000  10111000
 11100011  10111000  11101000  10111000  11101010  10111000  11101000  10111000
```

```
11101110  10111001
 11000010  10111000  11101011  10111000  11101000  10111000  11101011  00110000
10111001  11011100
 10111001  11000111  00110000  10111000  11101001  10111000  11101000  10111001
11000011  10111001
 11001010  10111000  11100111  10111001  11011101  10111000  11101011  10111001
11001101  10111000
 11100100  10111000  11100011  10111001  11000010  00111010  00110000  10111001
11000000  10111001
 11011101  10111000  11101110  10111000  11100011  10111001  11000010  00110000
10111000  11100010
 10111000  11101000  10111001  11000011  10111000  11100100  00111010  00110000
10111001  11000000
 10111000  11100001  10111000  11101110  10111000  11100100  10111000  11100011
10111001  11000010
 00111001



    String broken down to its symbols:

    [М] [и]
    [л] [ю] [б] [и] [м] [о]
    [ї] [с] [т] [и]
    [к] [а] [ш] [у] [!]
    [Т] [а] [к]
    [к] [а] [з] [а] [л] [и]
    [к] [о] [з] [а] [к] [и]
    [н] [а]
    [р] [у] [с] [і] [,]
    [д] [л] [я]
    [н] [и] [х]
    [т] [о] [й]
    [с] [м] [а] [к]
    [б] [у] [в]
    [н] [а] [й] [к] [р] [а] [щ] [и] [м] [,]
    [в] [і] [н]
    [н] [а] [г] [а] [д] [у] [в] [а] [в]
    [ї] [х]
    [б] [а] [т] [ь] [к] [і] [в] [щ] [и] [н] [у] [,]
    [р] [і] [д] [н] [у]
    [м] [а] [т] [и] [,]
    [р] [о] [д] [и] [н] [у] [.]

    CODEPOINTS SORTED:
strlen: 129
    1052    1080    32  1083    1102    1073    1080    1084    1086    32  1111
1089    1090    1080    32  1082    1072    1096    1091    33  32  1058    1072
1082    32  1082    1072    1079    1072    1083    1080    32  1082    1086    1079
1072    1082    1080    32  1085    1072    32  1088    1091    1089    1110    44  32
1076    1083    1103    32  1085    1080    1093    32  1090    1086    1081    32
```

```
1089    1084    1072    1082    32  1073    1091    1074    32  1085    1072    1081
1082    1088    1072    1097    1080    1084    44  32  1074    1110    1085    32
1085    1072    1075    1072    1076    1091    1074    1072    1074    32  1111
1093    32  1073    1072    1090    1100    1082    1110    1074    1097    1080
1085    1091    44  32  1088    1110    1076    1085    1091    32  1084    1072
1090    1080    44  32  1088    1086    1076    1080    1085    1091    46
    codepoint (32) was found -> 21
    codepoint (33) was found -> 1
    codepoint (44) was found -> 4
    codepoint (46) was found -> 1
    codepoint (1052) was found -> 1
    codepoint (1058) was found -> 1
    codepoint (1072) was found -> 14
    codepoint (1073) was found -> 3
    codepoint (1074) was found -> 5
    codepoint (1075) was found -> 1
    codepoint (1076) was found -> 4
    codepoint (1079) was found -> 2
    codepoint (1080) was found -> 10
    codepoint (1081) was found -> 2
    codepoint (1082) was found -> 8
    codepoint (1083) was found -> 3
    codepoint (1084) was found -> 4
    codepoint (1085) was found -> 8
    codepoint (1086) was found -> 4
    codepoint (1088) was found -> 4
    codepoint (1089) was found -> 3
    codepoint (1090) was found -> 4
    codepoint (1091) was found -> 7
    codepoint (1093) was found -> 2
    codepoint (1096) was found -> 1
    codepoint (1097) was found -> 2
    codepoint (1100) was found -> 1
    codepoint (1102) was found -> 1
    codepoint (1103) was found -> 1
    codepoint (1110) was found -> 4

    Chart of codepoints rarity:
    codepoint: [ 1103 ], count ->    1    [#]
    codepoint: [ 1097 ], count ->    2    [##]
    codepoint: [ 1089 ], count ->    3    [###]
    codepoint: [ 1110 ], count ->    4    [####]
    codepoint: [ 1074 ], count ->    5    [#####]
    codepoint: [ 1091 ], count ->    7    [#######]
    codepoint: [ 1085 ], count ->    8    [########]
    codepoint: [ 1080 ], count ->    10   [##########]
    codepoint: [ 1072 ], count ->    14   [##############]
    codepoint: [ 32 ], count ->  21   [######################]
```

```
+       String information:
|
|   Number of UTF-8 characters  : 129
|   Byte`s taken to store data  : 231
|   Bit length (bytes * 8)      : 1848
|   Unique characters count     : 31
|   Words in string             : 22
|   Etropy for the string       : 4.30
|   Kraft's value (UA+SYM = 38) : inf
|   Shannons value for UA_UTF8  : 0.39
+
```

Завдання 2

ПРИМІТКА

УВЕСЬ КОД ЩО ПОВ'ЯЗАНИЯ З UTF8 БУВ ПЕРЕНЕСЕНИЙ У ФАЙЛ "utf8.c" ДЛЯ ПОРТАТИВНОСТІ ТА СПРОЩЕННЯ ЧИТАЄМОСТІ САМОГО ЗАВДАННЯ

Кроки роботи программи:

1. Отримати строку з файлу
2. Вивести сторку на екран
3. Перетворити UTF-8 символи в числові коди
4. Створити "словник" для побудови дерева
5. Відсортувати словник
6. Вивести інформацію про строку до стиснення
7. Вивести вірогідності для появи кожного символу
8. Сторвити корінь дерева (пусте дерево)
9. Побудувати дерево
10. Побудувати масив кодів Шеннона-Фано для виведення на екран
11. Вивести інформацію після стиснненя (як в 6 пункті)
12. Декодувати коди назад в строку
13. Вивести строку
14. Звільнити ресурси (Heap пам'ять)

Код программи на C:

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```
#include <math.h>
#include "utf8.c"

// REQ:
//
// + Take string from first task
// + Shannons code
// + Check code by Shannons and Krafts method
// + Measure bitwidht
// + Average word encoding length
// + Coefficiency for compression and actual effectiveness
// + Decode result by using tree


// we store uinque character to encode
// with probability of it occurance in
// our string.

// #define DEBUG

#define iter(SIZE) for(uint i = 0; i<SIZE; i++)

typedef struct {
  uint  codepoint;
  float probability;
} Character;


typedef struct Node {
  enum { Left, Right , Top  } side           ;
  enum { Root, Branch, Leaf } type           ;
  Character*                  content        ;
  size_t                      content_size   ;
  struct Node*                next[2]        ;
} Node;


typedef struct {
  size_t     dictionary_size;
  Character* dictionary;
  Node       root;
} Tree;


// generic swap for numeric types
// void swap(float *xp, float *yp)
// {
//   float temp = *xp;
//   *xp = *yp;
//   *yp = temp;
// }
```

```
#define swap(A,B) _swap(&(A), &(B), sizeof(A))
void _swap(void * a, void * b, size_t len)
{
    unsigned char * p = a, * q = b, tmp;
    for (size_t i = 0; i != len; ++i)
    {
        tmp = p[i];
        p[i] = q[i];
        q[i] = tmp;
    }
}


void sort(uint* arr, int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}


void sort_dictionary(Character* arr, int n)
{
    int i, j, min_idx;

    for (i = 0; i < n-1; i++)
    {
        min_idx = i;
        for (j = i+1; j < n; j++)
          if (arr[j].probability < arr[min_idx].probability)
            min_idx = j;

        if(min_idx != i)
          swap(arr[min_idx], arr[i]);
    }
}


// BOILERPLATER
float self_information(float chance) {
  return ( -log2(chance) );
}
```

```
float shannons_value(float entropy, uint wordcount)
{
  return entropy / wordcount;
}

double krafts_value(uint symbols_count, uint alphabet_size)
{
  return pow(2, -symbols_count) * alphabet_size;
}



// parse string
// break it down to codepoints
// find all individual one
// get their probability
// create an array
// return it
Character* create_dictionary(uint* codepoints, size_t* dictionary_size, float
*entropy)
{
  // count total amount
  uint codepoints_count = 0;
  for(uint i = 0; codepoints[i] != ARRAY_END; i++)
    codepoints_count++;

  // copy codepoints
  uint* codepoints_buffer = malloc(sizeof(uint) * (codepoints_count+1) );
  for(uint i = 0; codepoints[i] != ARRAY_END; i++)
    codepoints_buffer[i] = codepoints[i];
  codepoints_buffer[codepoints_count] = ARRAY_END;

  // sort codepoints_buffer
  sort(codepoints_buffer, codepoints_count);


  // find max_count of codepoint
  uint cur_count = 0;
  uint max_count = 0;
  uint last_codepoint = 0;
  for(uint i = 0; codepoints_buffer[i] != 0; i++) {
    uint el = codepoints_buffer[i];

    if (cur_count > max_count)
      max_count = cur_count;

    if (el == last_codepoint)
      cur_count++;
    else {
      if (last_codepoint != 0) {
```

```
        // printf("\n\tcodepoint (%d) was found -> %d",last_codepoint,cur_count);
        float chance = (float)cur_count / (float)codepoints_count;
        (*entropy) +=  chance * self_information(chance);
      }
      last_codepoint = el;
      cur_count = 1;
    }
  }


  last_codepoint = 0;
  const uint EMPTY = 1;


  // create new Character entry for each unique character? lol
  //
  const size_t UNSET = 1;
  Character* dictionary = malloc(UNSET);

  for(uint i = 0; codepoints_buffer[i] != 0; i++) {
    uint el = codepoints_buffer[i];

    if (el == last_codepoint)
      cur_count++;
    else {

      if (last_codepoint != 0) {

        float probability =   (float)cur_count/(float)codepoints_count ;

        Character c = {last_codepoint, probability};
        (*dictionary_size)++;

        dictionary = realloc(dictionary, sizeof(Character) * (*dictionary_size) );
        dictionary[(*dictionary_size) - 1] = c;
        printf("\n\tcodepoint (%d) was found -> %d times",last_codepoint,cur_count);
      }

      last_codepoint = el;
      cur_count = 1;
    }
  }


  free(codepoints_buffer);
  return dictionary;
}

char* u8_encode_test(uint codepoint, uint nbyte) {

    char* dest = malloc(nbyte+1);
```

```
    memset(dest, 0, nbyte+1);
    uint8_t *byte = (uint8_t *)dest;

    switch (nbyte) {
        case 1:
            *byte = (uint8_t)codepoint;
            break;
        case 2:
            *byte = (uint8_t)(((codepoint >> 6) & HEAD_2BYTE_BITMASK) | 0xC0);
            byte++;
            *byte = (uint8_t)((codepoint & PAYLOAD_BITMASK) | 0x80);
            break;
        case 3:
            *byte = (uint8_t)(((codepoint >> 12) & HEAD_3BYTE_BITMASK) | 0xE0);
            byte++;
            *byte = (uint8_t)(((codepoint >> 6) & PAYLOAD_BITMASK) | 0x80);
            byte++;
            *byte = (uint8_t)((codepoint & PAYLOAD_BITMASK) | 0x80);
            break;
        case 4:
            *byte = (uint8_t)(((codepoint >> 18) & HEAD_4BYTE_BITMASK) | 0xF0);
            byte++;
            *byte = (uint8_t)(((codepoint >> 12) & PAYLOAD_BITMASK) | 0x80);
            byte++;
            *byte = (uint8_t)(((codepoint >> 6) & PAYLOAD_BITMASK) | 0x80);
            byte++;
            *byte = (uint8_t)((codepoint & PAYLOAD_BITMASK) | 0x80);
            break;
        default:
            printf("Invalid UTF-8 sequence\n");
            break;
    }
    return dest;
}


// Recursive function to find split_threshold based on probabilities
float split_index(Character* arr, int size) {
    float sum_left = 0.0f;
    float sum_right = 0.0f;
    int left_index = 0;
    int right_index = size - 1;
    float sum = 0;
    while (left_index <= right_index) {
        if (sum_left <= sum_right) {
            sum_left += arr[left_index].probability;
            left_index++;
        } else {
            sum_right += arr[right_index].probability;
            right_index--;
        }
```

```
    }

#ifdef DEBUG
    // Print the split
    printf("Left array:\n");
    for (int i = 0; i < left_index; i++) {
        printf("%.3f ", arr[i].probability);
    }
    printf("\nSum of left array: %.3f\n\n", sum_left);

    printf("Right array:\n");
    for (int i = right_index + 1; i < size; i++) {
        printf("%.3f ", arr[i].probability);
    }
    printf("\nSum of right array: %.3f\n", sum_right);
#endif

    for(uint i = 0; i < size; i++)
      if (sum == sum_left)
      {
        return i;
      }
      else
        sum += arr[i].probability;
    return 0;
}


void build_tree(Node* branch)
{

  Character *right_split, *left_split;
  uint split_i;
  float sum = 0;
  size_t left_size;
  size_t right_size;

  // find the center
  split_i= split_index(branch->content, branch->content_size);

  right_size = branch->content_size - split_i;
  left_size = branch->content_size - right_size;

  // Allocate splits

  right_split = malloc(sizeof(Character) * right_size);
  left_split  = malloc(sizeof(Character) * left_size );

  // set left
  for(uint i = 0; i < split_i; i++)
    // printf("child: %d,\tparent: %d\n",i,i);
```

```
    left_split[i] = branch->content[i];

  // set right
  for(uint i = split_i; i < branch->content_size; i++)
    // printf("child: %d,\tparent: %d\n",i-split_i,i);
    right_split[i-split_i] = branch->content[i];


  if (branch->type != Leaf) {

    // Create Left node
    Node* left_node = malloc(sizeof(Node));
    left_node->content_size = left_size;
    left_node->content      = left_split;
    left_node->side         = Left;
    if (left_size == 1)
      left_node->type       = Leaf;
    else
      left_node->type       = Branch;

    // Create Right node
    Node* right_node = malloc(sizeof(Node));
    right_node->content_size = right_size;
    right_node->content      = right_split;
    right_node->side         = Right;
    if (right_size == 1)
      right_node->type       = Leaf;
    else
      right_node->type       = Branch;

    // Set Next
    branch->next[0] = left_node ;
    branch->next[1] = right_node;

    build_tree(branch->next[0]);
    build_tree(branch->next[1]);
  }
  else {
    free(left_split);
    free(right_split);
  }

  // printf("left_s: %ld\t right_s: %ld\t\n\n",left_size,right_size);
}


void free_tree(Node* branch) {
    if (branch == NULL) {
        return;
    }
```

```
    if (branch->type != Leaf) {
        free_tree(branch->next[0]);
        free_tree(branch->next[1]);
    }
    free(branch->content); // Free content array of the current node
    free(branch); // Free the current node itself
}

void encode_with_tree(Node* node,uint codepoint, uint* depth, char* bytes_storage) {
    char byte;

    if (node->type == Leaf)
        goto end;

    byte = 1;
    for(uint i = 0; i < node->next[0]->content_size; i++)
        if (codepoint == node->next[0]->content[i].codepoint)
            if (node->next[0]->side == Left)
                byte = 0;
    sprintf(bytes_storage,"%s%d",bytes_storage,byte);
    (*depth)++;

    // printf("depth: %d\n",*depth);
    encode_with_tree(node->next[byte], codepoint, depth, bytes_storage);
end:;
}


int decode_with_tree(Node* n, char* sequence, uint current_pos) {

    int result;

    if (n->type == Leaf) {

#ifdef DEBUG
        printf("curr_pos: %s\t codepoints: [","-");
        for(uint i = 0; i < n->content_size;i++)
            printf("%d,",n->content[0].codepoint);
        printf("]\n");
#endif /* ifdef DEBUG */

        return n->content[0].codepoint;
    }

    char curr_pos = sequence[current_pos]-'0';

#ifdef DEBUG
    printf("curr_pos: %d\t codepoints: [",curr_pos);
    for(uint i = 0; i < n->content_size;i++)
        printf("%d,",n->content[i].codepoint);
    printf("]\n");
```

```
#endif /* ifdef DEBUG */

    result = decode_with_tree(n->next[curr_pos],sequence, current_pos+1);
    return result;
}




uint u8_words(const char* str){
  uint wc = 0;
  uint seen_letter = 0;

  for (uint i = 0; str[i] != 0; i++)
  {
    if (str[i] != ' ')
      seen_letter = 1;


    if (str[i] == ' '
        || str[i] == '\n'
        || str[i] == '\t'
        || str[i] == '\0'
        && seen_letter
       )
    {
      wc++;
    }
  }

  if (seen_letter && wc >= 1)
    wc ++;
  else if (seen_letter)
    wc ++;

  return wc;
}



size_t shannon_bits(char** compressed_sequence, size_t text_legnth)
{
    uint bits = 0;
    for (uint i = 0; i < text_legnth; i++)
        bits += strlen(compressed_sequence[i]);
    return bits;
}


//
//
```

```
// DEMONSTARTION
//
//
int main(int argc, char** argv)
{

  const uint UA_ALPHABET_SIZE    = 32 + 6;
  const uint UA_UTF8_SIZE        = 11; // 6 + 5 bytes
  float entropy                  = 0;
  size_t bitwidht                = 0;
  size_t bitwidht_compressed     = 0;
  size_t bytes                   = 0;
  size_t bytes_compressed        = 0;
  int    string_character_count  = 0;

  // load file
  char* str = load_string("file.txt");

  // transform to codepoints
  uint* codepoints_array = u8_as_codepoint_array(str);
  printf("\n\n\tString Original:\t`%s`\n\n",str);

  // create and sort dictionary from codepoints
  size_t dict_size = 0;
  Character* dictionary = create_dictionary(codepoints_array, &dict_size, &entropy);
  sort_dictionary(dictionary, dict_size);

  // complexity
  printf(
      "\n\n\n\n"
      "+\t\t" "String information uncompressed:" "\n"
      "|\t"   "\n"
      "|\t"   "Number of UTF-8 characters  : %d"       "\n"
      "|\t"   "Byte`s taken to store data  : %ld"      "\n"
      "|\t"   "Bit length (bytes * 8)      : %ld"      "\n"
      "|\t"   "Words in string             : %d"       "\n"
      "|\t"   "Etropy for the string       : %.2f"     "\n"
      "|\t"   "Kraft's value (UA+SYM = 38) : %.13lf"   "\n"
      "|\t"   "Shannons value for UA_UTF8  : %.2f"     "\n"
      "+\n"
      ,

      (string_character_count = u8strlen(str)   ),
      (bytes                  = strlen(str)     ),
      (bitwidht               = strlen(str)*8   ),
      u8_words(str),
      entropy,
      krafts_value(UA_UTF8_SIZE,UA_ALPHABET_SIZE),
      shannons_value(entropy, UA_UTF8_SIZE)
  );
  free(str);
```

```
  // print codepoints probabilities
  printf("\n\n\tcodepoints likehood: \n");
  for(uint i = 0; i<dict_size; i++)
    printf(
            "\t\t"
            "codepoint: %d" "\t"
            "probability: %f" "\n",
            dictionary[i].codepoint,dictionary[i].probability
  );

  // set up Tree Root
  Node* tree_root = malloc(sizeof(Node));
  tree_root->content_size = dict_size;
  tree_root->content = dictionary;
  tree_root->side = Top;
  tree_root->type = Root;

  // Build tree from the root
  build_tree(tree_root);

  // compress the message

  // len
  size_t text_legnth = 0;
  for(uint i = 0; codepoints_array[i] != ARRAY_END; i++)
      text_legnth++;

  // allocate message array
  const size_t MAX_TREE_DEPTH = 10;
  char** compressed_sequence = malloc(sizeof(char*) * (text_legnth) );

  // fill it up with compressed identifiers using Shannons tree
  for (uint i = 0; i < text_legnth; i++) {
      uint depth = 0;
      compressed_sequence[i] = calloc(MAX_TREE_DEPTH,sizeof(char) );
      encode_with_tree(tree_root, codepoints_array[i], &depth,
compressed_sequence[i]);
  }

  // print text string encoded
  printf(
          "\n\n\t Shennon-Fano coding for string: \n"
  );
  for (uint i = 0; i < text_legnth; i++) {
      if (i % 10 == 0)
          printf("\n\t\t");
      printf("[%s]\t",compressed_sequence[i]);
  }
```

```
  printf(
      "\n\n\n\n"
      "+\t\t" "String information COMPRESSED:" "\n"
      "|\t"   "\n"
      "|\t"   "Byte`s taken to store data  : %ld"      "\n"
      "|\t"   "Bit length (bytes * 8)      : %ld"      "\n"
      "|\t"   "Etropy for the string       : %.2f"     "\n"
      "|\t"   "Average bits per word       : %.2f"     "\n"
      "|\t"   "compression effectiveness   : %.2f"     "\t(does not count sizeof(tree)
)\n"
      "+\n"
      ,

      (shannon_bits(compressed_sequence, text_legnth) * 8),
      shannon_bits(compressed_sequence, text_legnth),
      entropy,
      (float)shannon_bits(compressed_sequence, text_legnth) / string_character_count,
      (double)bitwidht / shannon_bits(compressed_sequence, text_legnth)
  );

  // decode text back into codepoints
  uint* codepoints = calloc( (text_legnth+1), sizeof(uint));
  for (uint i = 0; i < text_legnth; i++) {
      const uint BEGINING = 0;
      codepoints[i] = decode_with_tree(
              tree_root,
              compressed_sequence[i],
              BEGINING
             );
  }

  printf(
          "\n\n\t Shennon-Fano DECODED: \n"
  );
  for (uint i = 0; i < text_legnth; i++) {
      if (i % 10 == 0)
          printf("\n\t\t");
      printf("[%d]\t",codepoints[i]);
  }

  // print string after Shannon compression and codepoints

  //   + back to stirng
  char* back = codepoint_array_as_u8str(codepoints);
  printf("\n\n\tString After compression and utf8 decoding:\n\t\t`%s`\n",back);


  // char* continer = calloc(10, 1);
  // uint depth = 0;
  // encode_with_tree(tree_root, 1090, &depth, continer);
  // uint cdp = decode_with_tree(tree_root, continer, 0);
```

```
  // printf("cdp: %d\n", cdp);

  // printf("Shannon code for 1090 in UTF-8:\t%s\n",continer);


  // defer block
  // free(continer);
  for(uint i = 0; i < text_legnth; i++)
      free(compressed_sequence[i]);


  free(codepoints);
  free(compressed_sequence);
  free_tree(tree_root);
  free(back);
  free(codepoints_array);
}
```

Результат роботи (збережений у файл використовуючи pipe operator (>))

```
clang task2.c -o task2 -g -O0 -fsanitize=address && ./task2 file.txt


    String Original:    `Ми любимо їсти кашу! Так казали козаки на русі, для них той
смак був найкращим, він нагадував їх батьківщину, рідну мати, родину.`


    codepoint (32) was found -> 21 times
    codepoint (33) was found -> 1 times
    codepoint (44) was found -> 4 times
    codepoint (46) was found -> 1 times
    codepoint (1052) was found -> 1 times
    codepoint (1058) was found -> 1 times
    codepoint (1072) was found -> 14 times
    codepoint (1073) was found -> 3 times
    codepoint (1074) was found -> 5 times
    codepoint (1075) was found -> 1 times
    codepoint (1076) was found -> 4 times
    codepoint (1079) was found -> 2 times
    codepoint (1080) was found -> 10 times
    codepoint (1081) was found -> 2 times
    codepoint (1082) was found -> 8 times
    codepoint (1083) was found -> 3 times
    codepoint (1084) was found -> 4 times
    codepoint (1085) was found -> 8 times
    codepoint (1086) was found -> 4 times
    codepoint (1088) was found -> 4 times
    codepoint (1089) was found -> 3 times
    codepoint (1090) was found -> 4 times
    codepoint (1091) was found -> 7 times
```

```
    codepoint (1093) was found -> 2 times
    codepoint (1096) was found -> 1 times
    codepoint (1097) was found -> 2 times
    codepoint (1100) was found -> 1 times
    codepoint (1102) was found -> 1 times
    codepoint (1103) was found -> 1 times
    codepoint (1110) was found -> 4 times



+       String information uncompressed:
|
|   Number of UTF-8 characters  : 129
|   Byte`s taken to store data  : 231
|   Bit length (bytes * 8)      : 1848
|   Words in string             : 22
|   Etropy for the string       : 4.30
|   Kraft's value (UA+SYM = 38) : inf
|   Shannons value for UA_UTF8  : 0.39
+


    codepoints likehood:
        codepoint: 33   probability: 0.007752
        codepoint: 46   probability: 0.007752
        codepoint: 1052 probability: 0.007752
        codepoint: 1058 probability: 0.007752
        codepoint: 1075 probability: 0.007752
        codepoint: 1096 probability: 0.007752
        codepoint: 1100 probability: 0.007752
        codepoint: 1102 probability: 0.007752
        codepoint: 1103 probability: 0.007752
        codepoint: 1079 probability: 0.015504
        codepoint: 1081 probability: 0.015504
        codepoint: 1093 probability: 0.015504
        codepoint: 1097 probability: 0.015504
        codepoint: 1083 probability: 0.023256
        codepoint: 1089 probability: 0.023256
        codepoint: 1073 probability: 0.023256
        codepoint: 1084 probability: 0.031008
        codepoint: 1086 probability: 0.031008
        codepoint: 1088 probability: 0.031008
        codepoint: 1090 probability: 0.031008
        codepoint: 44   probability: 0.031008
        codepoint: 1076 probability: 0.031008
        codepoint: 1110 probability: 0.031008
        codepoint: 1074 probability: 0.038760
        codepoint: 1091 probability: 0.054264
        codepoint: 1085 probability: 0.062016
        codepoint: 1082 probability: 0.062016
        codepoint: 1080 probability: 0.077519
```

```
        codepoint: 1072 probability: 0.108527
        codepoint: 32   probability: 0.162791


     Shennon-Fano coding for string:

        [0000001]   [101]   [111]   [001000]    [0000110]   [00101] [101]   [00110]
[00111] [111]
        [111]   [001001]    [01001] [101]   [111]   [1001]  [110]   [0000100]   [0111]
[00000000]
        [111]   [0000010]   [110]   [1001]  [111]   [1001]  [110]   [000100]    [110]
[001000]
        [101]   [111]   [1001]  [00111] [000100]    [110]   [1001]  [101]   [111]
[1000]
        [110]   [111]   [01000] [0111]  [001001]    [01100] [01010] [111]   [01011]
[001000]
        [0000111]   [111]   [1000]  [101]   [000110]    [111]   [01001] [00111]
[000101]    [111]
        [001001]    [00110] [110]   [1001]  [111]   [00101] [0111]  [01101] [111]
[1000]
        [110]   [000101]    [1001]  [01000] [110]   [000111]    [101]   [00110]
[01010] [111]
        [01101] [01100] [1000]  [111]   [1000]  [110]   [0000011]   [110]   [01011]
[0111]
        [01101] [110]   [01101] [111]   [111]   [000110]    [111]   [00101] [110]
[01001]
        [0000101]   [1001]  [01100] [01101] [000111]    [101]   [1000]  [0111]
[01010] [111]
        [01000] [01100] [01011] [1000]  [0111]  [111]   [00110] [110]   [01001] [101]
        [01010] [111]   [01000] [00111] [01011] [101]   [1000]  [0111]  [00000001]



+       String information COMPRESSED:
|
|   Byte`s taken to store data  : 4496
|   Bit length (bytes * 8)      : 562
|   Etropy for the string       : 4.30
|   Average bits per word       : 4.36
|   compression effectiveness   : 3.29  (does not count sizeof(tree) )
+


     Shennon-Fano DECODED:

        [1052]  [1080]  [32]    [1083]  [1102]  [1073]  [1080]  [1084]  [1086]  [32]
        [32]    [1089]  [1090]  [1080]  [32]    [1082]  [1072]  [1096]  [1091]  [33]
        [32]    [1058]  [1072]  [1082]  [32]    [1082]  [1072]  [1079]  [1072]  [1083]
        [1080]  [32]    [1082]  [1086]  [1079]  [1072]  [1082]  [1080]  [32]    [1085]
        [1072]  [32]    [1088]  [1091]  [1089]  [1110]  [44]    [32]    [1076]  [1083]
        [1103]  [32]    [1085]  [1080]  [1093]  [32]    [1090]  [1086]  [1081]  [32]
```

```
        [1089]  [1084]  [1072]  [1082]  [32]    [1073]  [1091]  [1074]  [32]    [1085]
        [1072]  [1081]  [1082]  [1088]  [1072]  [1097]  [1080]  [1084]  [44]    [32]
        [1074]  [1110]  [1085]  [32]    [1085]  [1072]  [1075]  [1072]  [1076]  [1091]
        [1074]  [1072]  [1074]  [32]    [32]    [1093]  [32]    [1073]  [1072]  [1090]
        [1100]  [1082]  [1110]  [1074]  [1097]  [1080]  [1085]  [1091]  [44]    [32]
        [1088]  [1110]  [1076]  [1085]  [1091]  [32]    [1084]  [1072]  [1090]  [1080]
        [44]    [32]    [1088]  [1086]  [1076]  [1080]  [1085]  [1091]  [46]

    String After compression and utf8 decoding:
        `Ми любимо  сти кашу! Так казали козаки на русі, для них той смак був
найкращим, він нагадував  х батьківщину, рідну мати, родину.`
```

Завдання 3

Примітка

УВЕСЬ КОД ЩО ПОВ'ЯЗАНИЯ З UTF8 БУВ ПЕРЕНЕСЕНИЙ У ФАЙЛ "utf8.c" ДЛЯ ПОРТАТИВНОСТІ ТА СПРОЩЕННЯ ЧИТАЄМОСТІ САМОГО ЗАВДАННЯ

Кроки роботи программи

Кроки роботи программи ідинтичні до таких в 2 роботі, за винятком того, що був змінена реалізація алгоритму, Алгоритм Huffman-а будує дерево з кінця, а не початку, для спрощення роботи, також було додано крок перетворення словника в список Node-ів.

Код програмии на C

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "utf8.c"

// REQ:
//
// + Take string from first task
// + Huffmans code
// + Check code by Shannons and Krafts method
// + Measure bitwidht
// + Average word encoding length
// + Coefficiency for compression and actual effectiveness
// + Decode result by using tree

// we store uinque character to encode
// with probability of it occurance in
```

```
// our string.

// HUFFMAN CODE
//
// huffman code does what Shannon-Fano codding does,
// but in reverse order, this means entierty of a encoding
// tree is build from leafs instead of the root.
//
// this change makes it MOST optimal compression alghorithm from
// math perspective.
//
// since such a tree is reverse order, it requires a minor
// tweaking from original Shannon-Fano implemetation
//


// #define DEBUG

#define iter(SIZE) for(uint i = 0; i<SIZE; i++)

#define BRANCH_CODEPOINT_FLAG -1
#define LEFT 0
#define RIGHT 1

typedef struct {
  uint  codepoint;
  float probability;
} Character;


typedef struct Node {
  uint  codepoint;
  float probability;
  struct Node* next[2];
} Node;


// generic swap for numeric types
// void swap(float *xp, float *yp)
// {
//   float temp = *xp;
//   *xp = *yp;
//   *yp = temp;
// }

#define swap(A,B) _swap(&(A), &(B), sizeof(A))
void _swap(void * a, void * b, size_t len)
{
    unsigned char * p = a, * q = b, tmp;
    for (size_t i = 0; i != len; ++i)
    {
```

```
        tmp = p[i];
        p[i] = q[i];
        q[i] = tmp;
    }
}


void sort(uint* arr, int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}



void sort_nodes(Node** arr, int n)
{
    int i, j, min_idx;

    for (i = 0; i < n-1; i++)
    {
        min_idx = i;
        for (j = i+1; j < n; j++)
          if (arr[j]->probability < arr[min_idx]->probability)
            min_idx = j;

        if(min_idx != i)
          swap(*arr[min_idx], *arr[i]);
    }

#ifdef DEBUG
    printf("\nNodes: \t[\t");
    for(uint i = 0; arr[i] != ARRAY_END; i++)
      printf(" {p: %f, c: %d}\t",arr[i]->probability,arr[i]->codepoint);
    printf("]\n");
#endif

}

void sort_dictionary(Character* arr, int n)
{
    int i, j, min_idx;
```

```
    for (i = 0; i < n-1; i++)
    {
        min_idx = i;
        for (j = i+1; j < n; j++)
          if (arr[j].probability < arr[min_idx].probability)
            min_idx = j;

        if(min_idx != i)
          swap(arr[min_idx], arr[i]);
    }
}


// BOILERPLATER
float self_information(float chance) {
  return ( -log2(chance) );
}

float shannons_value(float entropy, uint wordcount)
{
  return entropy / wordcount;
}

double krafts_value(uint symbols_count, uint alphabet_size)
{
  return pow(2, -symbols_count) * alphabet_size;
}



// parse string
// break it down to codepoints
// find all individual one
// get their probability
// create an array
// return it
Character* create_dictionary(uint* codepoints, size_t* dictionary_size, float
*entropy)
{
  // count total amount
  uint codepoints_count = 0;
  for(uint i = 0; codepoints[i] != ARRAY_END; i++)
    codepoints_count++;

  // copy codepoints
  uint* codepoints_buffer = malloc(sizeof(uint) * (codepoints_count+1) );
  for(uint i = 0; codepoints[i] != ARRAY_END; i++)
    codepoints_buffer[i] = codepoints[i];
  codepoints_buffer[codepoints_count] = ARRAY_END;
```

```
// sort codepoints_buffer
sort(codepoints_buffer, codepoints_count);


// find max_count of codepoint
uint cur_count = 0;
uint max_count = 0;
uint last_codepoint = 0;
for(uint i = 0; codepoints_buffer[i] != 0; i++) {
  uint el = codepoints_buffer[i];

  if (cur_count > max_count)
    max_count = cur_count;

  if (el == last_codepoint)
    cur_count++;
  else {
    if (last_codepoint != 0) {
      float chance = (float)cur_count / (float)codepoints_count;
      (*entropy) +=  chance * self_information(chance);
    }
    last_codepoint = el;
    cur_count = 1;
  }
}


last_codepoint = 0;
const uint EMPTY = 1;


// create new Character entry for each unique character? lol
//
const size_t UNSET = 1;
Character* dictionary = malloc(UNSET);

for(uint i = 0; codepoints_buffer[i] != 0; i++) {
  uint el = codepoints_buffer[i];

  if (el == last_codepoint)
    cur_count++;
  else {

    if (last_codepoint != 0) {

      float probability =   (float)cur_count/(float)codepoints_count ;

      Character c = {last_codepoint, probability};
      (*dictionary_size)++;

      dictionary = realloc(dictionary, sizeof(Character) * (*dictionary_size) );
```

```
        dictionary[(*dictionary_size) - 1] = c;
        printf("\n\tcodepoint (%d) was found -> %d times",last_codepoint,cur_count);
      }

      last_codepoint = el;
      cur_count = 1;
    }
  }


  free(codepoints_buffer);
  return dictionary;
}

char* u8_encode_test(uint codepoint, uint nbyte) {

    char* dest = malloc(nbyte+1);
    memset(dest, 0, nbyte+1);
    uint8_t *byte = (uint8_t *)dest;

    switch (nbyte) {
        case 1:
            *byte = (uint8_t)codepoint;
            break;
        case 2:
            *byte = (uint8_t)(((codepoint >> 6) & HEAD_2BYTE_BITMASK) | 0xC0);
            byte++;
            *byte = (uint8_t)((codepoint & PAYLOAD_BITMASK) | 0x80);
            break;
        case 3:
            *byte = (uint8_t)(((codepoint >> 12) & HEAD_3BYTE_BITMASK) | 0xE0);
            byte++;
            *byte = (uint8_t)(((codepoint >> 6) & PAYLOAD_BITMASK) | 0x80);
            byte++;
            *byte = (uint8_t)((codepoint & PAYLOAD_BITMASK) | 0x80);
            break;
        case 4:
            *byte = (uint8_t)(((codepoint >> 18) & HEAD_4BYTE_BITMASK) | 0xF0);
            byte++;
            *byte = (uint8_t)(((codepoint >> 12) & PAYLOAD_BITMASK) | 0x80);
            byte++;
            *byte = (uint8_t)(((codepoint >> 6) & PAYLOAD_BITMASK) | 0x80);
            byte++;
            *byte = (uint8_t)((codepoint & PAYLOAD_BITMASK) | 0x80);
            break;
        default:
            printf("Invalid UTF-8 sequence\n");
            break;
    }
    return dest;
}
```

```
Node** node_list_from_dict(Character* dictionary, size_t size) {
  Node** nl = malloc(sizeof(Node) * size);
  for(uint i = 0; i < size; i++) {
    nl[i] = malloc(sizeof(Node));
    nl[i]->probability = dictionary[i].probability;
    nl[i]->codepoint   = dictionary[i].codepoint;
    nl[i]->next[0] = NULL;
    nl[i]->next[1] = NULL;
  };
  return nl;
}

Node* build_tree(Node** node_list, size_t list_length)
{
  while(list_length != 1)
  {
    sort_nodes(node_list, list_length);

    Node* left =  node_list[LEFT]; // 0
    Node* right = node_list[RIGHT]; // 1

    Node *temp = malloc(sizeof(Node));
    temp->codepoint = BRANCH_CODEPOINT_FLAG;
    temp->probability = left->probability + right->probability;
    temp->next[LEFT] = left  ;
    temp->next[RIGHT] = right;

    node_list[1]  = node_list[list_length-1]; // copy last element to second index
(1)
    node_list[0]  = temp;                     // copy temp element into new address
(0)

    // printf("NODE: {p: %f, c: %d}\n"
    //     "\tchildren: [ {p: %f, c: %d} \t {p: %f, c: %d}  \n"
    //     "\n", temp->probability, temp->codepoint, left->probability, left-
>codepoint,
    //     right->probability, right->codepoint
    // );

    list_length--;
    node_list = realloc(node_list, sizeof(Node*) * list_length);
  }

  Node* pointer = node_list[0];
  free(node_list);

  return pointer;
```

```
}


void free_tree(Node* branch) {
    if (!branch)
      return;

    // printf("codepoint: %d\n", branch->codepoint);
    free_tree(branch->next[LEFT]);
    free_tree(branch->next[RIGHT]);
    free(branch); // Free the current node itself
}


void traverse_tree(Node* root, char* code, int index, int target_codepoint, char**
result) {
    if (root == NULL) return;

    // If the current node is a leaf node and matches the target codepoint
    if (root->codepoint == target_codepoint) {
        code[index] = '\0';  // Null terminate the string
        *result = strdup(code);  // strdup allocates memory for the string
        return;
    }

    // Traverse left
    code[index] = '0';
    traverse_tree(root->next[LEFT], code, index + 1, target_codepoint, result);

    // Traverse right
    code[index] = '1';
    traverse_tree(root->next[RIGHT], code, index + 1, target_codepoint, result);
}

char* get_codepoint_from_tree(Node* root, int target_codepoint) {
    if (root == NULL) return NULL;

    const size_t MAX_CODE_SIZE = 32;
    char* code = calloc(MAX_CODE_SIZE,sizeof(char));  // Assuming maximum code length
of 32
    char* result = NULL;

    traverse_tree(root, code, 0, target_codepoint, &result);
    free(code);  // Free temporary code array

    return result;
}

// void encode_with_tree(Node* node, uint codepoint, uint* depth, char* bytes_storage)
// {
//   if (!node)
```

```
//     return;
//
// }

// void encode_with_tree(Node* node,uint codepoint, uint* depth, char* bytes_storage)
{
//     char byte;
//
//     if (node->type == Leaf)
//         goto end;
//
//     byte = 1;
//     for(uint i = 0; i < node->next[0]->content_size; i++)
//         if (codepoint == node->next[0]->content[i].codepoint)
//             if (node->next[0]->side == Left)
//                 byte = 0;
//     sprintf(bytes_storage,"%s%d",bytes_storage,byte);
//     (*depth)++;
//
//     // printf("depth: %d\n",*depth);
//     encode_with_tree(node->next[byte], codepoint, depth, bytes_storage);
// end:;
// }
//
//
int decode_with_tree(Node* n, char* sequence, uint current_pos) {

    int result;

    if (n->codepoint != BRANCH_CODEPOINT_FLAG) {

#ifdef DEBUG
        printf("curr_pos: %s\t codepoints: [","-");
        for(uint i = 0; i < n->content_size;i++)
            printf("%d,",n->content[0].codepoint);
        printf("]\n");
#endif /* ifdef DEBUG */

        return n->codepoint;
    }

    if (!sequence) return -1;
    char curr_pos = sequence[current_pos]-'0';

#ifdef DEBUG
    printf("curr_pos: %d\t codepoints: [",curr_pos);
    for(uint i = 0; i < n->content_size;i++)
        printf("%d,",n->content[i].codepoint);
    printf("]\n");
#endif /* ifdef DEBUG */
```

```
    result = decode_with_tree(n->next[curr_pos],sequence, current_pos+1);
    return result;
}




uint u8_words(const char* str){
  uint wc = 0;
  uint seen_letter = 0;

  for (uint i = 0; str[i] != 0; i++)
  {
    if (str[i] != ' ')
      seen_letter = 1;


    if (str[i] == ' '
        || str[i] == '\n'
        || str[i] == '\t'
        || str[i] == '\0'
       && seen_letter
      )
    {
      wc++;
    }
  }

  if (seen_letter && wc >= 1)
    wc ++;
  else if (seen_letter)
    wc ++;

  return wc;
}



size_t shannon_bits(char** compressed_sequence, size_t text_legnth)
{
    uint bits = 0;
    for (uint i = 0; i < text_legnth; i++)
        bits += (compressed_sequence[i]) ? strlen(compressed_sequence[i]) : 1;
    return bits;
}


//
//
// DEMONSTARTION
//
```

```
//
int main(int argc, char** argv)
{

  const uint UA_ALPHABET_SIZE    = 32 + 6;
  const uint UA_UTF8_SIZE        = 11; // 6 + 5 bytes
  float entropy                  = 0;
  size_t bitwidht                = 0;
  size_t bitwidht_compressed     = 0;
  size_t bytes                   = 0;
  size_t bytes_compressed        = 0;
  int    string_character_count  = 0;

  // load file
  char* str = load_string("file.txt");

  // transform to codepoints
  uint* codepoints_array = u8_as_codepoint_array(str);
  printf("\n\n\tString Original:\t`%s`\n\n",str);

  // create and sort dictionary from codepoints
  size_t dict_size = 0;
  Character* dictionary = create_dictionary(codepoints_array, &dict_size, &entropy);
  sort_dictionary(dictionary, dict_size);

  // complexity
  printf(
      "\n\n\n\n"
      "+\t\t" "String information uncompressed:" "\n"
      "|\t"   "\n"
      "|\t"   "Number of UTF-8 characters  : %d"       "\n"
      "|\t"   "Byte`s taken to store data  : %ld"      "\n"
      "|\t"   "Bit length (bytes * 8)      : %ld"      "\n"
      "|\t"   "Words in string             : %d"       "\n"
      "|\t"   "Etropy for the string       : %.2f"     "\n"
      "|\t"   "Kraft's value (UA+SYM = 38) : %.13lf"   "\n"
      "|\t"   "Shannons value for UA_UTF8  : %.2f"     "\n"
      "+\n"
      ,

      (string_character_count = u8strlen(str)   ),
      (bytes                  = strlen(str)     ),
      (bitwidht               = strlen(str)*8   ),
      u8_words(str),
      entropy,
      krafts_value(UA_UTF8_SIZE,UA_ALPHABET_SIZE),
      shannons_value(entropy, UA_UTF8_SIZE)
  );
  free(str);
```

```
  // print codepoints probabilities
  printf("\n\n\tcodepoints likehood: \n");
  for(uint i = 0; i<dict_size; i++)
    printf(
            "\t\t"
            "codepoint: %d" "\t"
            "probability: %f" "\n",
            dictionary[i].codepoint,dictionary[i].probability
  );

  // set up Tree Root
  Node** nodelist = 0;
  // Build tree from the root
  nodelist        = node_list_from_dict(dictionary,dict_size);
  Node* tree_root = build_tree(nodelist,dict_size);


  // compress the message

  // len
  size_t text_legnth = 0;
  for(uint i = 0; codepoints_array[i] != ARRAY_END; i++)
      text_legnth++;

  // allocate message array
  const size_t MAX_TREE_DEPTH = 10;
  char** compressed_sequence = malloc(sizeof(char*) * (text_legnth) );

  // fill it up with compressed identifiers using Shannons tree
  for (uint i = 0; i < text_legnth; i++) {
      uint depth = 0;
      compressed_sequence[i] = get_codepoint_from_tree(tree_root,
codepoints_array[i]);
  }


  // print text string encoded
  printf(
          "\n\n\tstring: \n"
  );
  for (uint i = 0; i < text_legnth; i++) {
      if (i % 10 == 0)
          printf("\n\t\t");
      printf("[%d]\t",codepoints_array[i]);
  }

  // print text string encoded
  printf(
          "\n\n\t Shennon-Fano coding for string: \n"
  );
  for (uint i = 0; i < text_legnth; i++) {
```

```
      if (i % 10 == 0)
          printf("\n\t\t");
      printf("[%s]\t",compressed_sequence[i]);
  }

  printf(
      "\n\n\n\n"
      "+\t\t" "String information COMPRESSED:" "\n"
      "|\t"   "\n"
      "|\t"   "Byte`s taken to store data  : %ld"      "\n"
      "|\t"   "Bit length (bytes * 8)      : %ld"      "\n"
      "|\t"   "Etropy for the string       : %.2f"     "\n"
      "|\t"   "Average bits per word       : %.2f"     "\n"
      "|\t"   "compression effectiveness   : %.2f"     "\t(does not count sizeof(tree)
)\n"
      "+\n"
      ,

      (shannon_bits(compressed_sequence, text_legnth) * 8),
      shannon_bits(compressed_sequence, text_legnth),
      entropy,
      (float)shannon_bits(compressed_sequence, text_legnth) / string_character_count,
      (double)bitwidht / shannon_bits(compressed_sequence, text_legnth)
  );

  // decode text back into codepoints
  uint* codepoints = calloc( (text_legnth+1), sizeof(uint));
  for (uint i = 0; i < text_legnth; i++) {
    const size_t MAX_CODE_SIZE = 32;
    const uint BEGINING = 0;
    codepoints[i] = decode_with_tree(
        tree_root,
        compressed_sequence[i],
        BEGINING
    );
  }

  printf(
          "\n\n\t Shennon-Fano DECODED: \n"
  );
  for (uint i = 0; i < text_legnth; i++) {
      if (i % 10 == 0)
          printf("\n\t\t");
      printf("[%d]\t",codepoints[i]);
  }

  // print string after Shannon compression and codepoints

  //   + back to stirng
  char* back = codepoint_array_as_u8str(codepoints);
  printf("\n\n\tString After compression and utf8 decoding:\n\t\t`%s`\n",back);
```

```
  // char* continer = calloc(10, 1);
  // uint depth = 0;
  // encode_with_tree(tree_root, 1090, &depth, continer);
  // uint cdp = decode_with_tree(tree_root, continer, 0);
  // printf("cdp: %d\n", cdp);

  // printf("Shannon code for 1090 in UTF-8:\t%s\n",continer);


  // defer block
  // free(continer);
  for(uint i = 0; i < text_legnth; i++)
      free(compressed_sequence[i]);


  // free(codepoints);
  free(compressed_sequence);
  free(dictionary);
  free_tree(tree_root);
  // free_tree(tree_root);
  free(back);
  free(codepoints_array);
}
```

Результат роботи (збережений у файл використовуючи pipe operator (>))

```
clang task3.c -o task3 -g -O0 -fsanitize=address && ./task3 file.txt


    String Original:    `Ми любимо їсти кашу! Так казали козаки на русі, для них той
смак був найкращим, він нагадував їх батьківщину, рідну мати, родину.`


    codepoint (32) was found -> 21 times
    codepoint (33) was found -> 1 times
    codepoint (44) was found -> 4 times
    codepoint (46) was found -> 1 times
    codepoint (1052) was found -> 1 times
    codepoint (1058) was found -> 1 times
    codepoint (1072) was found -> 14 times
    codepoint (1073) was found -> 3 times
    codepoint (1074) was found -> 5 times
    codepoint (1075) was found -> 1 times
    codepoint (1076) was found -> 4 times
    codepoint (1079) was found -> 2 times
    codepoint (1080) was found -> 10 times
    codepoint (1081) was found -> 2 times
    codepoint (1082) was found -> 8 times
```

```
    codepoint (1083) was found -> 3 times
    codepoint (1084) was found -> 4 times
    codepoint (1085) was found -> 8 times
    codepoint (1086) was found -> 4 times
    codepoint (1088) was found -> 4 times
    codepoint (1089) was found -> 3 times
    codepoint (1090) was found -> 4 times
    codepoint (1091) was found -> 7 times
    codepoint (1093) was found -> 2 times
    codepoint (1096) was found -> 1 times
    codepoint (1097) was found -> 2 times
    codepoint (1100) was found -> 1 times
    codepoint (1102) was found -> 1 times
    codepoint (1103) was found -> 1 times
    codepoint (1110) was found -> 4 times



+       String information uncompressed:
|
|   Number of UTF-8 characters  : 129
|   Byte`s taken to store data  : 231
|   Bit length (bytes * 8)      : 1848
|   Words in string             : 22
|   Etropy for the string       : 4.30
|   Kraft's value (UA+SYM = 38) : inf
|   Shannons value for UA_UTF8  : 0.39
+


    codepoints likehood:
        codepoint: 33   probability: 0.007752
        codepoint: 46   probability: 0.007752
        codepoint: 1052 probability: 0.007752
        codepoint: 1058 probability: 0.007752
        codepoint: 1075 probability: 0.007752
        codepoint: 1096 probability: 0.007752
        codepoint: 1100 probability: 0.007752
        codepoint: 1102 probability: 0.007752
        codepoint: 1103 probability: 0.007752
        codepoint: 1079 probability: 0.015504
        codepoint: 1081 probability: 0.015504
        codepoint: 1093 probability: 0.015504
        codepoint: 1097 probability: 0.015504
        codepoint: 1083 probability: 0.023256
        codepoint: 1089 probability: 0.023256
        codepoint: 1073 probability: 0.023256
        codepoint: 1084 probability: 0.031008
        codepoint: 1086 probability: 0.031008
        codepoint: 1088 probability: 0.031008
        codepoint: 1090 probability: 0.031008
```

```
        codepoint: 44   probability: 0.031008
        codepoint: 1076 probability: 0.031008
        codepoint: 1110 probability: 0.031008
        codepoint: 1074 probability: 0.038760
        codepoint: 1091 probability: 0.054264
        codepoint: 1085 probability: 0.062016
        codepoint: 1082 probability: 0.062016
        codepoint: 1080 probability: 0.077519
        codepoint: 1072 probability: 0.108527
        codepoint: 32   probability: 0.162791


    string:

        [1052]  [1080]  [32]    [1083]  [1102]  [1073]  [1080]  [1084]  [1086]  [32]
        [1111]  [1089]  [1090]  [1080]  [32]    [1082]  [1072]  [1096]  [1091]  [33]
        [32]    [1058]  [1072]  [1082]  [32]    [1082]  [1072]  [1079]  [1072]  [1083]
        [1080]  [32]    [1082]  [1086]  [1079]  [1072]  [1082]  [1080]  [32]    [1085]
        [1072]  [32]    [1088]  [1091]  [1089]  [1110]  [44]    [32]    [1076]  [1083]
        [1103]  [32]    [1085]  [1080]  [1093]  [32]    [1090]  [1086]  [1081]  [32]
        [1089]  [1084]  [1072]  [1082]  [32]    [1073]  [1091]  [1074]  [32]    [1085]
        [1072]  [1081]  [1082]  [1088]  [1072]  [1097]  [1080]  [1084]  [44]    [32]
        [1074]  [1110]  [1085]  [32]    [1085]  [1072]  [1075]  [1072]  [1076]  [1091]
        [1074]  [1072]  [1074]  [32]    [1111]  [1093]  [32]    [1073]  [1072]  [1090]
        [1100]  [1082]  [1110]  [1074]  [1097]  [1080]  [1085]  [1091]  [44]    [32]
        [1088]  [1110]  [1076]  [1085]  [1091]  [32]    [1084]  [1072]  [1090]  [1080]
        [44]    [32]    [1088]  [1086]  [1076]  [1080]  [1085]  [1091]  [46]

     Shennon-Fano coding for string:

        [1001110]   [1101]  [111]   [00010] [11001111]  [00100] [1101]  [10000]
[10001] [111]
        [(null)]    [00011] [10111] [1101]  [111]   [0111]  [010]   [1001101]   [0011]
[1001000]
        [111]   [1001111]   [010]   [0111]  [111]   [0111]  [010]   [100101]    [010]
[00010]
        [1101]  [111]   [0111]  [10001] [100101]    [010]   [0111]  [1101]  [111]
[0110]
        [010]   [111]   [10110] [0011]  [00011] [11000] [10100] [111]   [10101]
[00010]
        [1100110]   [111]   [0110]  [1101]  [001011]    [111]   [10111] [10001]
[001010]    [111]
        [00011] [10000] [010]   [0111]  [111]   [00100] [0011]  [0000]  [111]   [0110]
        [010]   [001010]    [0111]  [10110] [010]   [110010]    [1101]  [10000]
[10100] [111]
        [0000]  [11000] [0110]  [111]   [0110]  [010]   [1001100]   [010]   [10101]
[0011]
        [0000]  [010]   [0000]  [111]   [(null)]    [001011]    [111]   [00100] [010]
[10111]
        [11001110]  [0111]  [11000] [0000]  [110010]    [1101]  [0110]  [0011]
[10100] [111]
```

```
        [10110] [11000] [10101] [0110]  [0011]  [111]   [10000] [010]   [10111] [1101]
        [10100] [111]   [10110] [10001] [10101] [1101]  [0110]  [0011]  [1001001]



+       String information COMPRESSED:
|
|   Byte`s taken to store data  : 4456
|   Bit length (bytes * 8)      : 557
|   Etropy for the string       : 4.30
|   Average bits per word       : 4.32
|   compression effectiveness   : 3.32  (does not count sizeof(tree) )
+


     Shennon-Fano DECODED:

        [1052]  [1080]  [32]    [1083]  [1102]  [1073]  [1080]  [1084]  [1086]  [32]
        [-1]    [1089]  [1090]  [1080]  [32]    [1082]  [1072]  [1096]  [1091]  [33]
        [32]    [1058]  [1072]  [1082]  [32]    [1082]  [1072]  [1079]  [1072]  [1083]
        [1080]  [32]    [1082]  [1086]  [1079]  [1072]  [1082]  [1080]  [32]    [1085]
        [1072]  [32]    [1088]  [1091]  [1089]  [1110]  [44]    [32]    [1076]  [1083]
        [1103]  [32]    [1085]  [1080]  [1093]  [32]    [1090]  [1086]  [1081]  [32]
        [1089]  [1084]  [1072]  [1082]  [32]    [1073]  [1091]  [1074]  [32]    [1085]
        [1072]  [1081]  [1082]  [1088]  [1072]  [1097]  [1080]  [1084]  [44]    [32]
        [1074]  [1110]  [1085]  [32]    [1085]  [1072]  [1075]  [1072]  [1076]  [1091]
        [1074]  [1072]  [1074]  [32]    [-1]    [1093]  [32]    [1073]  [1072]  [1090]
        [1100]  [1082]  [1110]  [1074]  [1097]  [1080]  [1085]  [1091]  [44]    [32]
        [1088]  [1110]  [1076]  [1085]  [1091]  [32]    [1084]  [1072]  [1090]  [1080]
        [44]    [32]    [1088]  [1086]  [1076]  [1080]  [1085]  [1091]  [46]

    String After compression and utf8 decoding:
        `Ми любимо сти кашу! Так казали козаки на русі, для них той смак був
найкращим, він нагадував х батьківщину, рідну мати, родину.`
```

ВИСНОВКИ

При виконанні цієї Лабораторної роботи я ознайомився з фундаметальними принципами внутрішньої роботи найпопулярнішого формату кодування строк, зрозумів на яких фундаментальних приницпах працює ПЗ для стискання даних в архівах, та загалом покращив своє розуміння у роботі стандартних бібліотек сучасних мов програмування, та внутрішній принцип роботи та кодування в байтах пам'яті комп'ютера. Також я покращив свої уміння у написанні программ на мові програмування C.

PS

Якщо ви вже помітили, при декодуванні строки в кодові значення (будь-то HEX чи DECIMAL), по якійсь причині буква "ї" на відміну від усіх інших символів, або не

> паристься, або не включається у фінальний словник. Для мене це стало помітним тільки у 3 роботі коли я її уже формально закінчив. Я спробував знайти корінь проблеми, але через нестачу часу та дедлайн завдання в мене не було нагоди вирішити цю проблему :x .

Якщо ви прочитали звіт повністю, то..

ДЯКУЮ ЗА УВАГУ! :)