

# 확률 및 통계

Assignment #4: Entropy Coding – Huffman Encoder, Decoder

학 과: 컴퓨터정보공학부

담당교수님: 심동규 교수님

학 번: 2015722068

이 름: 남윤창

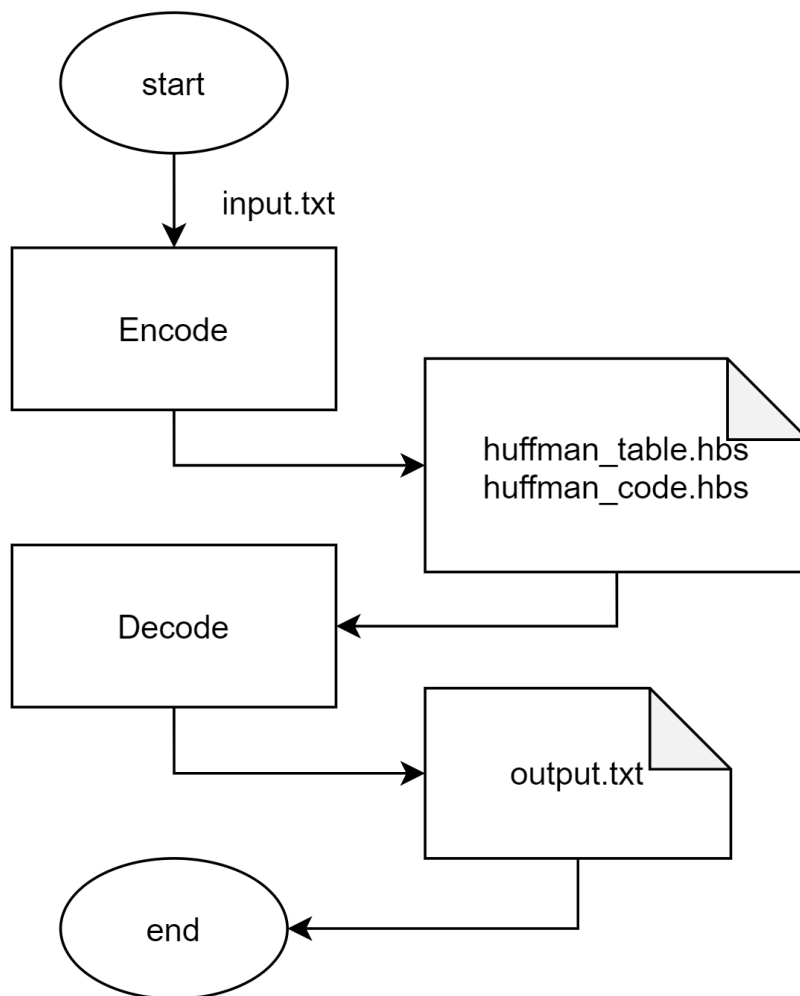
## 1. Introduction

본 과제는 ASCII code와 Huffman code를 서로 변환하는 Huffman Encoder와 Decoder를 작성한다. Huffman Encoder는 input.txt를 읽어 빈도를 체크하여 그에 맞게 허프만 테이블을 생성한다. 이를 비트스트림으로 변환하여 huffman\_table.hbs를 생성한다. 이를 이용하여 huffman\_code.hbs에 binary file로 각각 저장한다. Decoder는 앞서 생성한 두 바이너리 파일을 이용하여 허프만 테이블을 역으로 생성하고 이를 이용하여 원본으로 복구하여 output.txt로 저장한다.

ASCII code중 안 쓰는 것을 임의로 정의하여 end of data로 지정하여 파일의 끝을 인식하게 한다. 이는 인코더, 디코더 서로 임의로 약속한다고 가정한다.

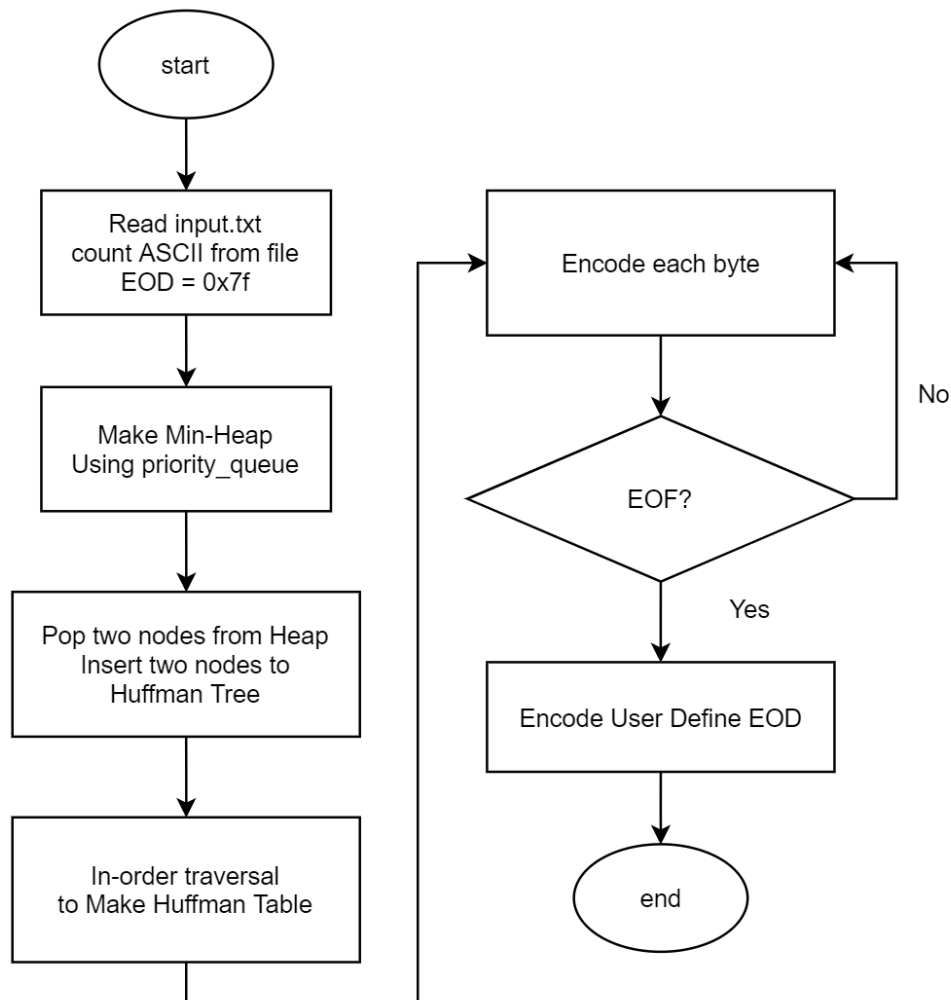
1byte씩 읽기 때문에 마지막 eod를 넣고 공간이 남는다면 왼쪽으로 shift하고 나머지를 0으로 채우는 byte align을 고려한다.

이를 통해 인코딩 된 Huffman\_code.hbs가 기존 input.txt에 비해 얼마나 사이즈가 줄었는지 확인한다. 디코딩 된 결과 output.txt가 input.txt와 사이즈가 같게 되었는지 비교한다.



## 2. Flow chart & Algorithms

### # Encoder



#### Step 1.

Encoder의 전반적인 Flow chart는 위와 같다. Input.txt 파일을 읽어 1byte씩 ASCII값을 읽는다. 이때 End of file은 0x7f로 지정한다. 각 ASCII값을 읽어 개수를 count한 다음 이를 HuffmanTreeNode 구조체를 통해 관리한다.

#### Step 2.

이를 이용해 Min-Heap 자료구조를 생성한다. Count한 ASCII의 빈도수를 이용하여 정렬한다. 가장 작은 값이 항상 root에 있도록 하기 위해 Min-Heap 자료구조를 사용하였고 C++ STL의 priority queue를 이용했다. 가장 빈도 수가 적은 두개의 노드를 Pop하여 HuffmanTree에 insert한다. 두 개의 빈도수를 더한 값을 갖는 새로운 부모 노드를 생성하여 pop된 두 노드를 왼쪽 오른쪽으로 각각 하나씩 갖게 한다. 그리고 새로 생성된 부모 노드를 다시 Min-Heap에 push한다. 이 과정을 Min-Heap의 원소가 1개만 남을 때까지

반복한다. 이후 마지막 남은 원소는 자연스레 HuffmanTree의 root가 되도록 한다.

### Step 3.

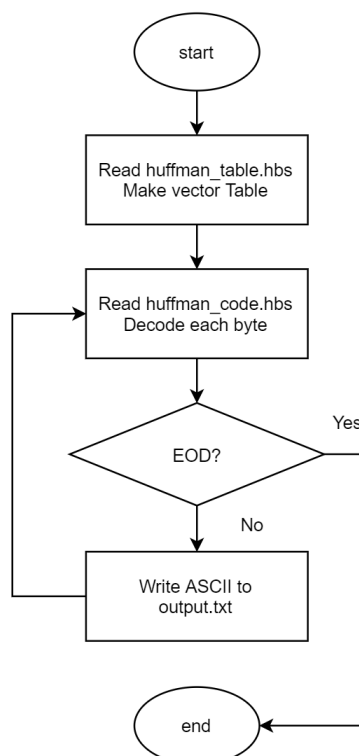
생성된 HuffmanTree를 In-order traversal하여 각 ASCII에 대해 Encoded bit를 부여한다. 이는 재귀함수로 구현하였으며 left child node일 경우 bit 0을 붙여주고 right child node일 경우 1을 붙여준다. Leaf node일 경우만 붙여주고 나머지 부모노드의 경우 실제 값이 있는 것이 아니므로 skip하도록 구현했다.

각각 ASCII마다 생성된 bit을 통해 Huffman Table을 생성한다. string으로 붙어있는 bit들을 실제 bit단위로 바꿔주는 작업을 진행한다. 이 과정은 source code 부분에서 코드와 함께 추가 설명한다.

### Step 4.

Input.txt파일을 다시 읽는다. 각 ASCII값을 미리 생성해둔 Huffman\_table.hbs에서 찾아서 Huffman\_code.hbs파일에 저장한다. 저장할 때는 8bit, 즉 1byte씩 저장해야 하므로 계속 읽어 8bit가 채워지면 저장하도록 구현했다. 자세한 내용은 Source code 부분에 설명되어 있다. 파일의 마지막이 된다면 미리 정의해둔 EOD값을 임의로 넣는다. 이 경우 EOD를 포함하여 마지막 bit까지 8의 배수라면 상관없지만 아닐 경우 bit수가 맞지 않으므로 EOD를 왼쪽으로 채우고 뒤에 0으로 채우도록 구현하였다. 이 부분에서 stuffing과 byte align을 고려하여 구현하였다.

### # Decoder



Step 1.

디코딩을 하기 위해서는 인코딩 된 테이블과 코드 파일, Huffman\_table.hbs, Huffman\_code.hbs가 필요하다. 우선 Huffman\_table.hbs를 읽는다. 인코딩을 할 경우 in-order traversal로 저장하였기 때문에 순서대로 읽어도 in-order순으로 새롭게 저장된다. 따라서 위와 같이 순서대로 정보를 HuffmanTreeNode 구조체에 저장하고 이를 vector배열에 차례대로 저장한다. vector에 push하면 bitstream 순서로 vector를 순회하며 정보를 읽으므로 자연스럽게 Tree를 읽는 순서가 된다.

Step 2.

Huffman\_code.hbs를 읽어서 미리 저장된 vector에서 해당 bitstream이 있는지 검사하여 ASCII값으로 변환하여 output.txt에 저장한다. 인코딩에서 미리 약속해둔 EOD값을 만나게 되면 디코딩을 중지하도록 하였다. Bitstream을 읽어서 변환하는 과정은 Source code에서 부가 설명한다.

### 3. Source Code

# Encoder.cpp

Step 1.

```
// Step 1. Load File
FILE * input = fopen("input.txt", "r");
unsigned char chTemp;
int ascii[128] = { 0, }; //count ASCII
if (!input)
    printf("파일 안열림 \n");
// 아스키 카운트
while (1)
{
    chTemp = fgetc(input);
    if (feof(input)) {
        break;
    }
    ascii[chTemp]++;
}
// Step 1.1 End Of Data 설정
unsigned char end_of_data = 0x7f;
ascii[end_of_data]++;

// 허프만 트리 노드 구조체
typedef struct HuffmanTreeNode {
    string encoded; //인코딩 된 수
    int freq; //빈도수
    unsigned char ascii_code; //ASCII CODE
    HuffmanTreeNode* leftnode; //왼쪽 노드
    HuffmanTreeNode* rightnode; //오른쪽 노드
} HuffmanTreeNode;
```

다음과 같이 ASCII를 카운팅 하고 EOD를 설정한다.

```
// Step 2. Min-Heap에 Insert. Used Priority_queue
priority_queue< pair<int, HuffmanTreeNode*>, vector< pair<int, HuffmanTreeNode*> >, greater< pair<int, HuffmanTreeNode*> > > pq;
for (int i = 0; i < 128; i++) {
    if (ascii[i] != 0) {
        HuffmanTreeNode *t;
        t = new HuffmanTreeNode; //노드 설정
        t->ascii_code = i;
        t->freq = ascii[i];
        t->leftnode = NULL;
        t->rightnode = NULL;
        pq.push(make_pair(ascii[i], t));
    }
}
```

이를 pair 자료형으로 priority\_queue를 통해 관리한다. 모든 정보를 priority\_queue에 push하여 Min-Heap을 구현한다.

Step 2.

```
// 허프만 트리 삽입
 HuffmanTreeNode* HuffmanTree_Insert(HuffmanTreeNode* u, HuffmanTreeNode* v) {
    HuffmanTreeNode *pt;
    pt = new HuffmanTreeNode;    //부모 노드 설정
    pt->ascii_code = NULL;
    pt->freq = u->freq + v->freq;    //빈도수를 합한다.
    pt->leftnode = u;                //좌, 우 노드 설정
    pt->rightnode = v;
    return pt;    //부모노드 반환
}
```

노드 2개를 pop해서 HuffmanTree에 insert한다. U, v는 각 노드이며 해당 노드의 frequency를 더해서 부모노드를 만든다. 부모노드의 ASCII값은 0으로 주어 의미 없는 값이라는 것을 판별해준다. U,v는 각각 좌, 우 자식노드로 설정한다.

Step 3.

```
// 허프만 트리 중위순회
void Inorder_traversal(HuffmanTreeNode* t, string num, int &tnum) {
    if (t != NULL) {
        Inorder_traversal(t->leftnode, num + "0", tnum);
        if (t->ascii_code != NULL) {
            t->encoded = num;
            cout << " " << t->ascii_code << "Wt" << t->freq << "Wt"
                << t->encoded << " " << endl;
            tnum = MakeTable(t, tnum);
        }
        Inorder_traversal(t->rightnode, num + "1", tnum);
    }
}
```

중위순회를 재귀로 구현하였다. string자료형을 사용하여 + suffix연산자를 통하여 bitstream을 만들었다. 이를 Huffman Table에 저장하도록 구현했다.

Table은 정보가 다음과 같이 저장된다.

0101010	00000010	N	8 - N
---------	----------	---	-------

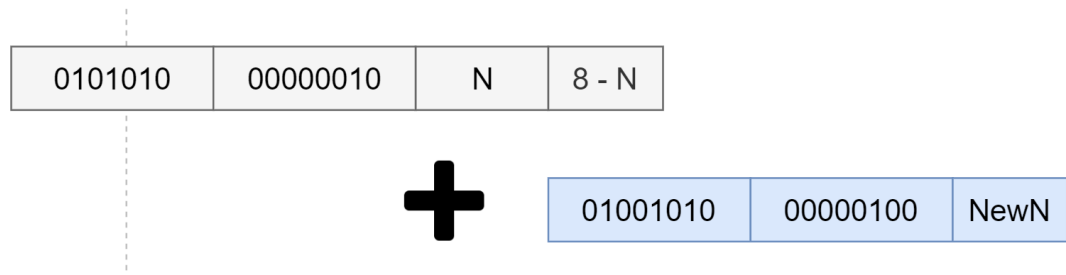
앞의 8bit은 ASCII값, 중간 8bit은 codeword의 length, 그리고 length만큼의 bit를 N이라

고 할 경우 N bit만큼 codeword가 저장된다. 이후 새로운 ASCII 정보를 table에 저장하기 위해서는 8-N 자리에 새로 작성해야 한다. 따라서 File Pointer와 Nbit에 대한 정보를 꾸준히 프로그램은 기억하고 있어야한다. Table을 만드는 코드는 다음과 같다.

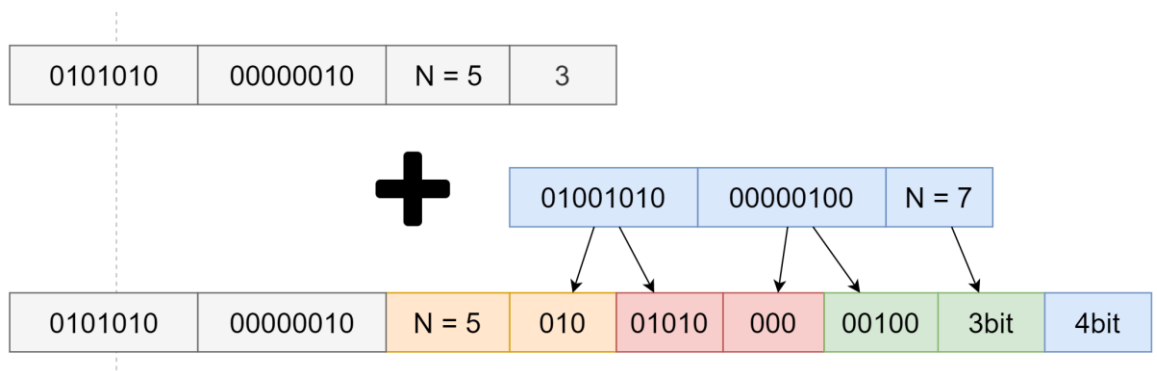
N이 0인 경우는 자연스럽게 8, 8, N만큼 file에 출력하면 되므로 설명은 생략하고 N이 0이 아닌 경우이다.

```
temp_c = new unsigned char[bit_length + 1];
strcpy((char *)temp_c, t->encoded.c_str()); //string to unsigned char arr
if (tnum != 0) {
    unsigned char ascii_hi = t->ascii_code >> tnum;
    unsigned char ascii_lo = t->ascii_code << (8 - tnum); //ascii 자름
    unsigned char bit_length_hi = bit_length >> tnum;
    unsigned char bit_length_lo = bit_length << (8 - tnum); //bitlength 자름
    fseek(huff_fp, 0, SEEK_END);
    fseek(huff_fp, -1L, SEEK_CUR);
}
```

변수 tnum은 위 설명의 N의 역할이다. 따라서 ASCII값을 N에 맞게 뒤에 이어 붙이기 위해 tnum을 사용해서 shift하는 모습이다. 그림으로 자세히 설명한다.



N이 아닌 경우는 이어 붙여야 하므로 8bit를 쪼개서 bit단위로 연산을 해줄 수밖에 없다 따라서 코드와 마찬가지로 N(tnum)으로 shift를 해준다. 각각 shift한 결과를 hi, lo로 이름을 붙여주었다.

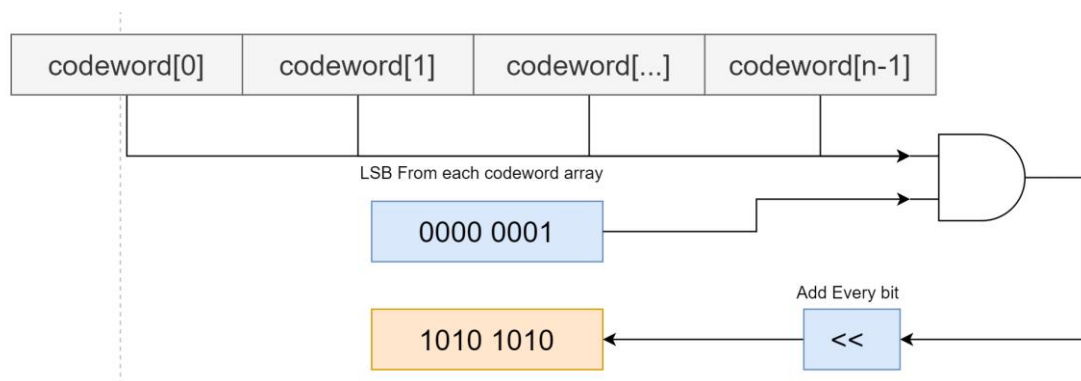


위의 경우 N(tnum)이 5인 경우 새로운 값을 tnum만큼 shift하여 기존 공간에 이어 붙이고 남은 bit를 8-tnum하여 새로운 공간에 저장한다. 뒤에 bit length값도 똑같이 tnum과 8-tnum으로 각각 shift하여 저장하고 codeword도 마찬가지로 shift한다. 같은 색깔의 block은 같은 1byte로 저장이 되었음을 뜻한다.

Codeword에 대한 추가 설명이다. HuffmanTreeNode에는 codeword를 string형태로 저장하였다. 하지만 위에서 설명한 내용과 같이 encoding하려면 string을 bit단위로 바꿔줘야 한다. 따라서 다음과 같이 코딩했다.

```
int codeword_count = 0;
for (int i = 0; i < 8 - tnum; i++) { //codeword 앞부분
    if (temp_c[codeword_count] == NULL) {
        end_flag = true;
        break;
    }
    if (temp_c[codeword_count] & 0x01 == 0x01) {
        bit_length_lo += 0x01 << (7 - tnum - i);
    }
    else {
        bit_length_lo += 0x00 << (7 - tnum - i);
    }
    codeword_count++;
}

int temp_cc = codeword_count;
if (end_flag == false) {
    unsigned char codeword_lo = NULL;
    for (int j = 0; j < (bit_length - temp_cc - 1) / 8 + 1; j++) { //codeword
        codeword_lo = NULL;
        if (temp_c[codeword_count] == NULL) break;
        for (int i = 0; i < 8; i++) {
            if (temp_c[codeword_count] == NULL) break;
            if (temp_c[codeword_count] & 0x01 == 0x01) {
                codeword_lo += 0x01 << (7 - i);
            }
            else {
                codeword_lo += 0x00 << (7 - i);
            }
            codeword_count++;
        }
        fprintf(huff_fp, "%c", codeword_lo);
    }
}
```



String을 unsigned char 배열로 변환한다. 이후 각 배열의 LSB에 원하는 정보가 담겨 있기 때문에 0x01을 이용하여 AND연산 한 후 배열의 index에 따라 shift하여 새로운 8bit character정보를 제작하도록 구현했다.

Step 4.

위와 같이 생성한 Table을 이용하여 Encoding을 진행하였다.

```
// Step 5. Encoding
input = fopen("input.txt", "r");
if (!input)
    printf("파일 안열림 \n");
tnum = 0;
while (1)
{
    chTemp = fgetc(input);
    if (feof(input)) {
        GetFromTable(end_of_data, tnum);
        break;
    }
    GetFromTable(chTemp, tnum);
}

fclose(input);
```



Input.txt를 다시 열어서 처음부터 읽으면서 Table에서 정보를 읽어 character단위로 파일에 출력한다. GetFromTable의 내용은 아래와 같다.

```
if (ascii_table == ch) { //찾던 결과
    int code_cnt = bl_table;
    for (int i = 0; i < arr_size; i++) { //get codeword
        unsigned char temp_cw = NULL;
        if (tnum + code_cnt > 8) { ... }
        else { //안빠져나간경우
            temp_cw = fgetc(table_fp) << tnum;
            temp_cw = (temp_cw & (0xff << (8 - code_cnt))); //다 왼쪽으로 채움
            codeword[i] += temp_cw;
            last_bit_length = code_cnt;
        }
    }
    Encoding(codeword, arr_size, wr_tnum, last_bit_length); //파일 출력
    break;
}
```

Table에서 내가 찾던 결과가 나올 때까지 반복하고 내가 찾던 결과가 나온다면 현재 Huffman\_code.hbs의 마지막 bit 상황에 따라 shift를 진행해주고 파일에 출력한다. 앞서 설명한 그림과 동일하다. 파일의 마지막 유효 비트를 N(tnum)이라고 할 경우 bit length + tnum이 8보다 큰 경우 잘라서 넣게 된다. 그리고 tnum값을 업데이트 한다. 이 과정은 Encoding함수에 있다.

마지막에 해당하는 EOD까지 Huffman\_code.hbs에 출력하면 Encoding 프로그램은 끝나게 된다.

# Decoder.cpp

```
int main() {
    vector<HuffmanTreeNode*> table;
    // Step 1. huffman_table.hbs를 갖고 table을 만든다.
    MakeTableFromEncoded(table);

    int cnt = 0;
    while (1) {
        cout << " " << table[cnt]->ascii_code << "Wt" //vector 확인
              << table[cnt]->encoded << " " << endl; //inorder로 저장되어있음
        if (table.back() == table[cnt]) {
            break;
        }
        cnt++;
    }

    // Step 2. vector huffman table을 통하여 decode
    unsigned char end_of_data = 0x7f;
    Decode(table, end_of_data);
}
```

Decoder.cpp의 main이다. 크게 2개의 step으로 나누었다. Table을 만드는 과정과 이를 해독하는 과정이다.

Step 1.

Encoding에서 설명한 바와 마찬가지로 유효 비트수를 N으로 하고 이에 따라 ASCII값, Bit length, codeword를 shift연산과 file pointer를 옮겨가며 Huffman\_table.hbs에서 불러온다. 이는 그대로 HuffmanTreeNode 구조체에 저장된다.

```
// HuffmanTreeNode에 값을 넣어주는 함수
void InsertNode(vector<HuffmanTreeNode*> &v, int size,
    unsigned char * cw, unsigned char lbl, unsigned char ascii){
    string result;
    HuffmanTreeNode *t;
    t = new HuffmanTreeNode;
    t->ascii_code = ascii;
    for (int i = 0; i < size; i++) {
        if (i == size - 1) {
            UCharToString(result, cw[i], lbl);
        }
        else {
            UCharToString(result, cw[i], 8);
        }
    }
    t->encoded = result;
    v.push_back(t);
}
```

각각 쉬프트 연산을 통해 codeword, codeword배열의 size, 마지막 codeword배열의 length, 그리고 그에 해당하는 ASCII값을 인자로 넣어 HuffmanTreeNode를 구축한다. 인코딩과 다른 점은 여기서는 빈도수가 중요하지 않기 때문에 빈도수에 대한 내용은 없는 노드가 생성된다.

HuffmanTreeNode의 codeword는 string형태이므로 bit단위의 codeword를 읽어서 string으로 바꾸어 주어야 한다. 이에 해당하는 함수는 UCharToString으로 코드는 다음과 같다.

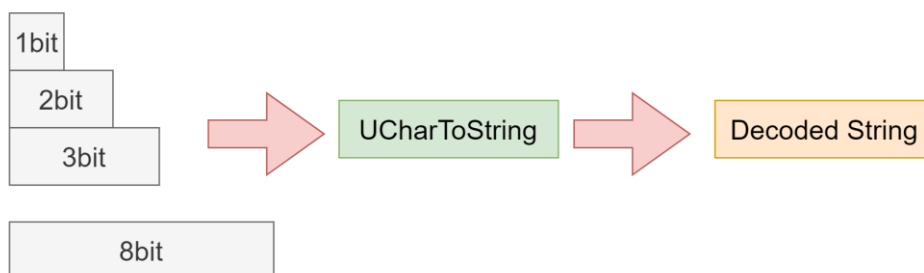
```
// unsigned char 를 string으로 bit단위 읽어가며 바꾸는 함수
void UCharToString(string &result, unsigned char ch, unsigned char length) {
    for (int j = 7; j > 7 - length; j--) {
        if ((0x01 & (ch >> j)) == 0x01) {
            result += '1';
        }
        else {
            result += '0';
        }
    }
}
```

앞서 인코딩에서 했던 내용을 거꾸로 하면 된다. 유효비트 length만큼 반복하여 0x01과 shift and연산을 통해 1이면 1, 0이면 0으로 suffix해준다.

Step 2.

```
while (1) {
    code_hi = fgetc(code_fp) << tnum;
    code_lo = fgetc(code_fp) >> (8 - tnum);
    fseek(code_fp, -1L, SEEK_CUR);
    code = code_hi + code_lo;
    for (int i = 7; i >= 0; i--) { //8번 돌려서 Search값, 원하는값 나올때까지 반복
        temp_s = s + "";
        UCharToString(temp_s, code, 8 - i);
        decoded = SearchVector(v, temp_s); //찾는 값이 없음
        if (decoded == eod) { //EOD읽음
            return;
        }
        if (decoded != NULL) {
            s = "";
            temp_s = "";
            find_flag = true;
            WriteASCII(decoded);
            tnum += (8 - i);
            fseek(code_fp, -1L, SEEK_CUR); // N -= 1
        }
    }
    if (tnum >= 8) {
        fseek(code_fp, 1L, SEEK_CUR);
        tnum -= 8;
    }
    else break;
}
break;
}
if (find_flag == false) {
    s += temp_s;
}
find_flag = false;
}
fclose(code_fp);
```

Decoding 코드이다. 디코딩은 Huffman\_code.hbs를 읽어서 앞서 생성한 vector를 순회한다. Character를 읽어서 1bit씩 늘려보며 string으로 변환하고 이에 해당하는 내용이 Vector에 있는지 검사한다. 만약 찾는 값이 있다면 이를 읽어와 output.txt에 저장한다.



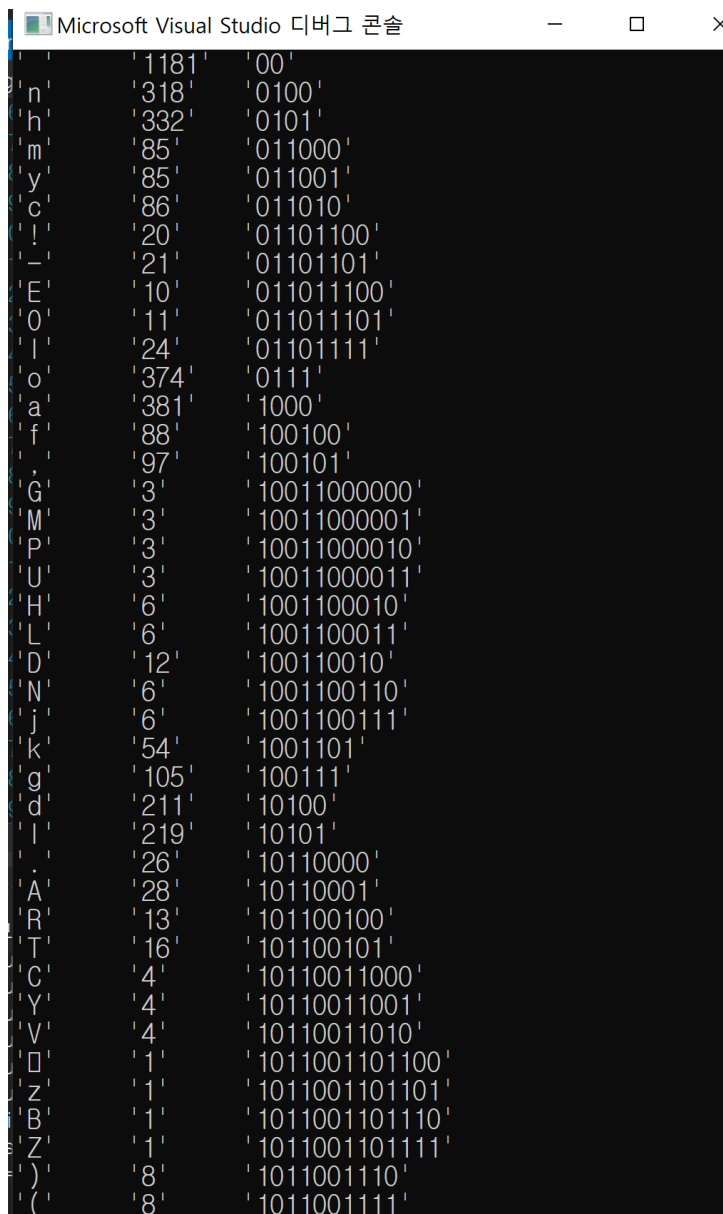
다음과 같이 1bit부터 8bit까지 모두 넣어보며 찾는다고 보면 된다. 만약 8bit까지 해도

없다면 내용을 저장하고 다시 다음 1byte를 읽어 9bit~~ 로 쭉 진행하게 된다. 이미 해당하는 값으로 Table을 만들었기 때문에 무조건 String이 matching이 될 수밖에 없다. 만약 찾지 못한다면 Huffman\_table.hbs 가 손상을 입은 경우일 것이다.

마지막으로 EOD를 읽는다면 Write하지 않고 바로 디코딩이 종료된다.

#### 4. Result

# Encoding 결과



```
Microsoft Visual Studio 디버그 콘솔
n      1181  '00'
h      318  '0100'
m      332  '0101'
y      85   '011000'
c      85   '011001'
!      86   '011010'
_      20   '01101100'
-      21   '01101101'
E      10   '011011100'
O      11   '011011101'
l      24   '01101111'
o      374  '0111'
a      381  '1000'
f      88   '100100'
.      97   '100101'
G      3    '10011000000'
M      3    '10011000001'
P      3    '10011000010'
U      3    '10011000011'
H      6    '1001100010'
L      6    '1001100011'
D      12   '100110010'
N      6    '1001100110'
j      6    '1001100111'
k      54   '1001101'
g      105  '100111'
d      211  '10100'
i      219  '10101'
.      26   '10110000'
A      28   '10110001'
R      13   '101100100'
T      16   '101100101'
C      4    '10110011000'
Y      4    '10110011001'
V      4    '10110011010'
□      1    '1011001101100'
z      1    '1011001101101'
B      1    '1011001101110'
Z      1    '1011001101111'
)      8    '1011001110'
(      8    '1011001111'
```

제공된 input.txt를 encoding한 결과이다 위 console은 encoding을 in-order traversal 하며 출력한 내용이고 순서대로 ASCII, 빈도수, bitstream이다.

Visual Studio Binary File 편집기를 사용하였다.

huffman_table.hbs	huffman_code.hbs	encoder.cpp
00000000 20 02 1B 81 11 A0 11 5B 41 98 79 06 65 8C 19 A2		.....[A.y.e...
00000010 10 86 C2 D0 86 D4 50 96 E2 78 4B 75 24 21 BD BC		.....P...xKu\$!...
00000020 11 D8 41 21 98 1A 42 C0 69 51 C2 E6 02 68 5C C1		..A!...B.iQ...hW.
00000030 50 0B 98 4A A1 73 0D 20 2A 62 4C 0A 98 D1 02 66		P...J.s. *bL...f
00000040 49 C1 53 33 50 54 CE D6 0F 35 9C 1A 76 40 5A 36		l.S3PT...5...v@Z6
00000050 02 D4 B8 22 C1 04 22 C5 48 26 C8 A8 13 65 43 0B		...".H&...eC.
00000060 B3 0B 21 76 65 58 2E CD 3F 86 D9 B1 E8 36 CD A8		..!veX...?....6..
00000070 41 B6 6E 5A 0D B3 79 48 55 9C 50 15 67 85 03 5A		A.nZ...yHU.P.g..Z
00000080 E4 0B 77 40 4C 75 06 D1 C0 1F 53 B0 46 B2 58 66		..w@Lu....S.F.Xf
00000090 B8 23 06 6B 8B 88 66 B9 3C 06 6B 99 D8 56 BA C0		..#..k...f.<..k..V..
000000a0 13 AF 73 05 DB 28 27 34 82 F9 DC 1B E2 70 8F C3		..s..('4.....p..
000000b0 A0 AF D1 5C 2B F5 53 0A FD 8F C2 BF 76 20 7F E0		...W+.S.....v ..
000000c0		


Huffman\_tabel.hbs 결과이다. 처음에 Spacebar이 저장되었는데 이에 대한 값이 0x20임을 보아 제대로 저장되었음을 미루어 볼 수 있다. Decoding 과정에서 추가로 분석한다.

huffman_table.hbs	huffman_code.hbs	encoder.cpp
00000c20 F9 28 7B 0F 97 90 DA F1 9A 5D 3A 86 2F 5A D3 DD		..({.....]:./Z..
00000c30 3A F2 9F F4 C3 17 8B 24 7F FF EC 3E 8D 88 E5 5D		.....\$.>...]
00000c40 27 EB 98 E7 FE 37 DC 9F A0 6D 78 91 F4 4A 5A BD		'...7...mx...JZ.
00000c50 7D F5 64 3C 84 15 74 9E 52 AF F0 B1 5A CA 7C BE		}..d<...t.R...Z.. .
00000c60 69 4F A3 65 7D 86 9A 8F EC 90 5D FE 1E 42 B0 C6		i0.e}.....]..B..
00000c70 AD DA B0 93 F9 27 24 BB B0 62 F1 77 79 2C 2D B4		.....'\$..b.wy,-.
00000c80 2C AB D7 E3 EE BF 14 77 BE C8 AD 4B BE 89 43 17		.....w...K..C.
00000c90 85 8A D6 53 FE 98 62 F3 27 DD 7E 22 B5 2A ED 4D		...S..b..'~"*..M
00000ca0 EA 6B B6 C2 50 F9 0B 1A F9 AE 16 28 7F EE 42		..k..P.y.....(..B
00000cb0 2B 53 17 8F A1 92 8F F2 0E 9C 6F D4 E2 12 86 9A		+S.....o.....
00000cc0 8C 5E B5 F1 7A F2 99 76 7C 93 9D AD D7 64 A3 BD		..^...z..v ....d..
00000cd0 E5 36 BC 7D 15 9B D4 37 14 AB 25 1F E4 31 78 63		..6.}...7...%..1xc
00000ce0 D4 A5 7A 5B 7E 74 A7 5F C9 38 AF F1 B5 E3 E8 D9		..z[~t...8.....
00000cf0 DA DD 73 1C 9F B0 7D 30 89 E3 C9 61 6D A1 FB 69		..s...}0...am...i
00000d00 4A 72 55 93 6B C3 50 C7 1A 6A 74 20 AF B3 2B C6		JrU.k.P...jt...+.
00000d10 2D FB 9B 6B D3 CF D4 32 3F D7 A5 22 B5 18 8A 70		-..k...2?..."..p
00000d20 F2 5B B7 AF A8 4F 58 DE F5 D3 17 AF C7 D1 B1 1F		..[...0X.....
00000d30 17 C9 38 E8 7B 0E D7 36 BB 56 10 67 91 92 77 AD		..8.{...6.V.g...w.
00000d40 39 09 BC CC B6 C2 50 B1 AF 9A EF CD 92 7A FB C3		9.....P.....z..
00000d50 15 F4 9D 70 FA 36 62 C6 1E C1 8F 4E B8 7F EA BA		...p.6b...N...
00000d60 4E 63 8E 9C 3C 86 2F 5B 47 7B EC 79 0C 5E 16 2B		Nc...<./[G{.y.^.+
00000d70 5D 74 FF A6 4A 45 63 6D 87 7B 17 AE 62 F1 57 6A		]t...JEcm.{...b.Wj
00000d80 6E CF BA FC 63 B9 58 BC FD 28 F7 B7 17 89 BC C9		n...c.X..(.....
00000d90 F4 6C C7 73 6C 45 6B 29 FF 4C 23 08 46 4B C6 71		..l..sIEk).L#.FK.q
00000da0 EC 3E 7D 2B 41 1F 07 D5 C8 42 32 3C 96 E2 F3 16		..>}+A....B2<....
00000db0 01 31 3F DD AD D7 94 74 31 78 DF 34 E9 43 3C C7		..1?...t1x.4.C<..
00000dc0 17 7D 12 92 9B 5E 1A 86 38 D3 53 A1 05 5F EB 5A		..}...^...8.S...Z
00000dd0 D2 F9 1E 43 6B C2 C5 04 7C 11 F3 CD 75 0F FD 23		...Ck... ...u...#
00000de0 DF A5 21 28 7F E5 F2 50 F6 1F 46 C8 2B EC CA F5		..!(...P...F.+...
00000df0 B4 77 B9 1F DF 4C 24 F4 99 DC 87 91 A5 ED 8B E9		..w...L\$.....
00000e00 D7 F4 0D AF 19 7F 75 0C 5E 2B EC CA F1 3B D6 9C		.....u.^+....;
00000e10 84 DE 66 DF 21 8B C5 5D A9 B2 90 94 31 C5 EB 93		..f.!...]....1...
00000e20 DF A3 0A 75 7D 3A E1 EC 24 F6 67 51 B2 CD 80		...u}:...\$.gQ...


Huffman\_code.hbs 결과이다. 메모리의 0부터 0x0e30까지 약 4KB정도를 차지한 것을 볼 수 있다. 인코딩이 제대로 된 것은 Decoding과정에서 추가로 분석한다.

input 속성

일반 보안 자세히 이전 버전

input

파일 형식: 텍스트 문서(.txt)

연결 프로그램:  메모장 

변경(C)...

위치: C:\Users\Wnyc09\source\repos\Wassignme


크기: 6.39KB (6,551 바이트)

디스크 할당 크기: 8.00KB (8,192 바이트)


압축에 사용된 input.txt는 정확히 6551 Byte이다.

huffman\_code 속성

일반 보안 자세히 이전 버전

huffman\_code

파일 형식: HBS 파일(.hbs)

연결 프로그램:  메모장 

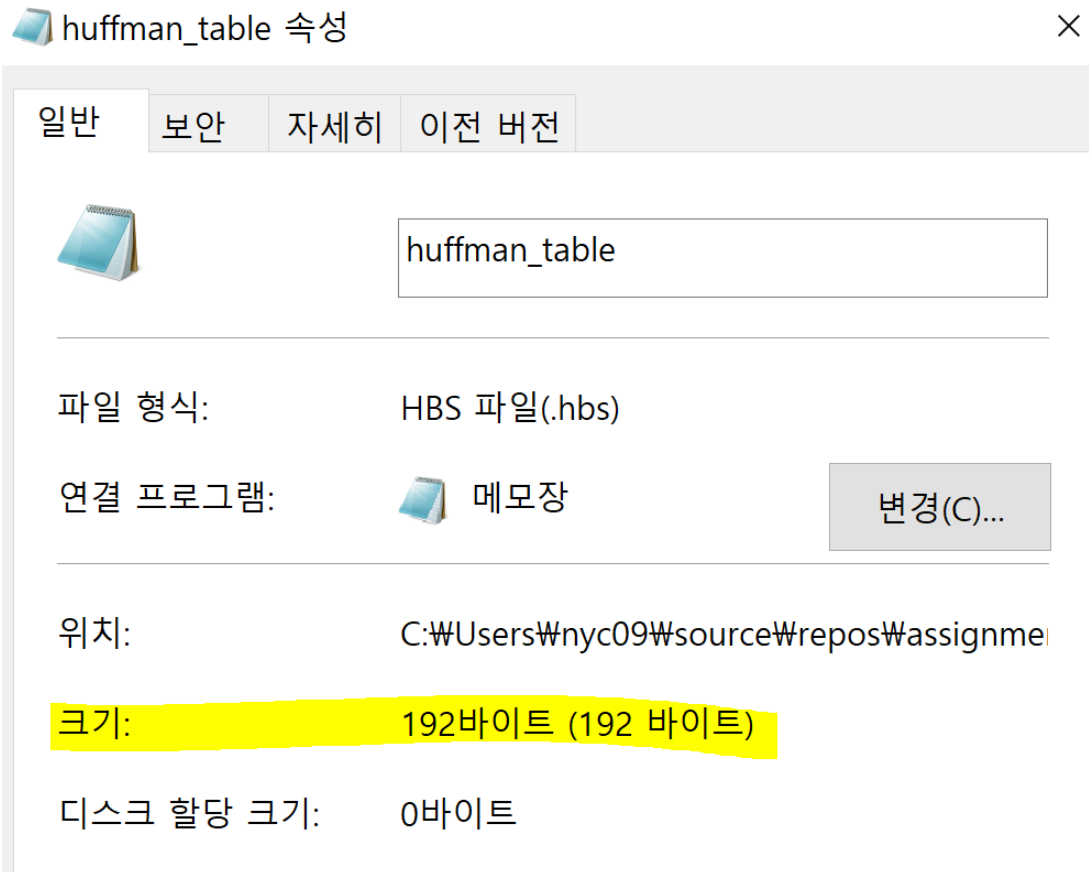
변경(C)...

위치: C:\Users\Wnyc09\source\repos\Wassignme

크기: 3.54KB (3,631 바이트)

디스크 할당 크기: 4.00KB (4,096 바이트)

압축된 결과 Huffman\_code.hbs는 3631 byte이다.



압축된 Huffman\_table.hbs는 192byte이다.

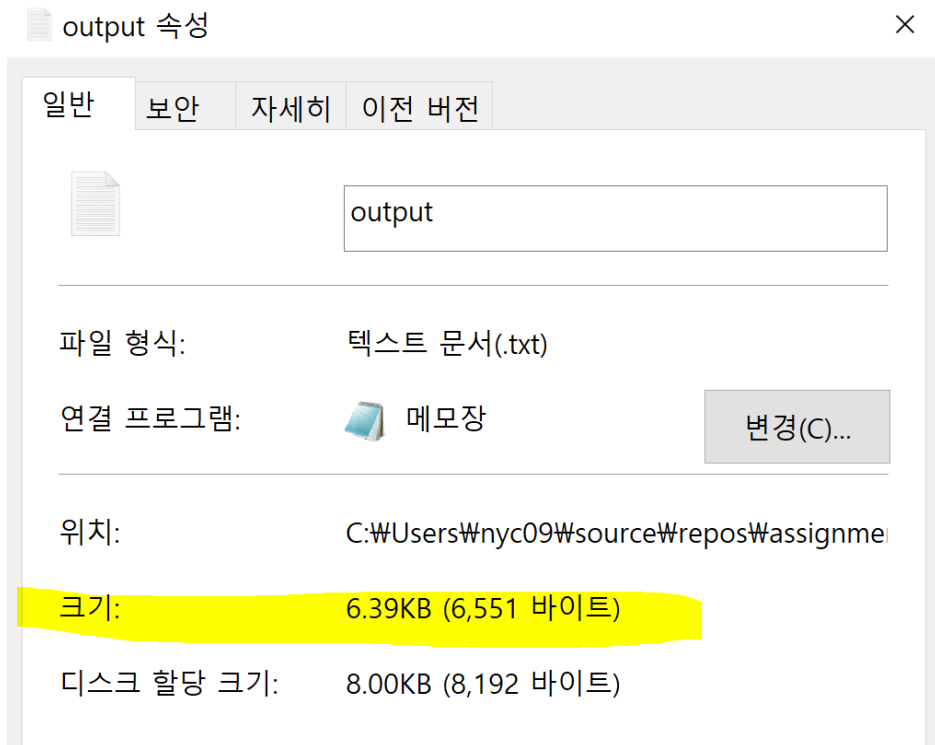
따라서 원래 Data는 6551 byte, 압축 이후 data의 크기는  $3631 + 192$  byte이다.

인코딩 테이블을 무시하고 Huffman\_code.hbs만 본다면 크기가 원래 크기에 비해 55.4267% 정도가 되었음을 확인할 수 있다. 약 45% 압축이 된 것이다.

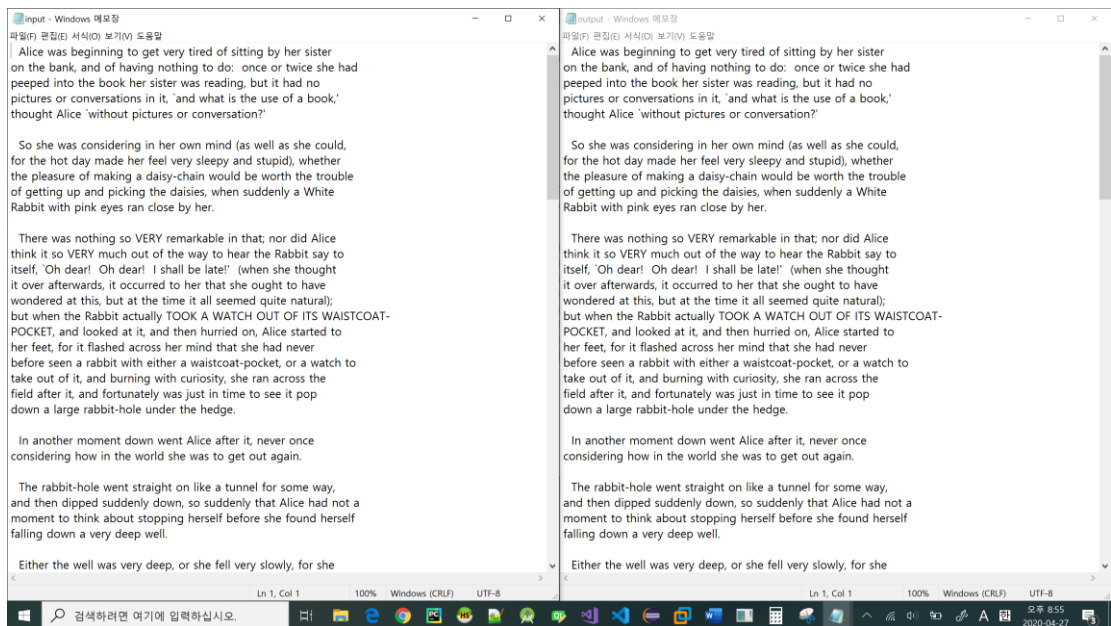
인코딩 테이블을 포함한다면 원래 크기에 비해 58.3575% 정도가 되었음을 확인할 수 있다. 약 42% 압축이 된 것이다.

통계적으로 자세한 내용은 고찰에 추가 설명하였다.

## # Decoding 결과



우선 디코딩 된 파일의 크기는 6551 byte로 입력에 사용된 input.txt와 byte수가 똑 같은 것을 확인하여 디코딩이 제대로 진행되었음을 확인했다.



<좌: input.txt / 우: output.txt>



해당 text파일이 오류 없이 인코딩 -> 디코딩 되었음을 확인했다.

온라인에서 텍스트를 비교하여 다른점을 찾아주는 프로그램에 올려보았을 경우도 다음과 같이 다른점이 없음을 확인했다.

```
There were doors all round the hall, but they were all locked;  
and when Alice had been all the way down one side and up the  
other, trying every door, she walked sadly down the middle,  
wondering how she was ever to get out again.  
  
Suddenly she came upon a little three-legged table, all made of  
solid glass; there was nothing on it except a tiny golden key,  
and Alice's first thought was that it might belong to one of the  
doors of the hall; but, alas! either the locks were too large, or  
the key was too small, but at any rate it would not open any of  
them. However, on the second time round, she came upon a low  
curtain she had not noticed before, and behind it was a little  
door about fifteen inches high: she tried the little golden key  
in the lock, and to her great delight it fitted!
```

Microsoft Visual Studio 디버그 콘솔

```
'00'  
'n' '0100'  
'h' '0101'  
'm' '011000'  
'y' '011001'  
'c' '011010'  
'!' '01101100'  
'-' '01101101'  
'E' '011011100'  
'O' '011011101'  
'l' '01101111'  
'o' '0111'  
'a' '1000'  
'f' '100100'  
'.' '100101'  
'G' '10011000000'  
'M' '10011000001'  
'P' '10011000010'  
'U' '10011000011'  
'H' '1001100010'  
'L' '1001100011'  
'D' '100110010'  
'N' '1001100110'  
'j' '1001100111'  
'k' '1001101'  
'g' '100111'  
'd' '10100'  
'l' '10101'  
'.' '10110000'  
'A' '10110001'  
'R' '101100100'  
'T' '101100101'  
'C' '10110011000'  
'Y' '10110011001'  
'V' '10110011010'  
'Q' '1011001101100'  
'z' '1011001101101'  
'B' '1011001101110'  
'Z' '1011001101111'  
')' '1011001110'  
'(' '1011001111'  
  
'r' '101101'  
't' '10111'  
'u' '1100'  
'p' '110100'  
'v' '1101010'  
'K' '11010110'  
'110101110000'
```

또한 디코딩 된 Table의 결과는 console에 출력하여 ASCII와 bitstream이 일치하는지 확인했다.

## 5. Consideration

본 과제는 Huffman Entropy 개념에 기반한 Encoding, Decoding 프로그램을 C, C++로 작성하는 것이다. 기본적으로 빈도수가 높은 ASCII에 짧은 bitstream을 부여하고 빈도수가 낮은 ASCII에 높은 bit length를 부여한다. 위의 결과화면에서 확인할 수 있듯이 띄어쓰기는 1181번 나와 가장 높은 비중을 차지하고 2bit만 차지하게 된다. 반대로 EOD의 경우 딱 1번 나온다. 이는 무려 13bit의 길이를 갖는다.

ASCII는 각각 8bit를 갖는다. 이를 Entropy 다음 entropy 공식에 대입하게 된다면 Entropy의 값은 8이된다. 모든 bit이 8bit으로 mapping되어있기 때문이다.

$$E(x) = \sum_i p_i \log_2 \frac{1}{p_i}$$

하지만 Huffman Table에 따라 코딩을 하게 된다면 각 bit가 8bit고정이 아니라 빈도수가 많은 경우 8bit보다 적게 갖고 빈도수가 높은 것은 8bit보다 높게 갖게 된다. 해당 input.txt에 대해 실제로 entropy를 측정해보았다.

```
int main() {
    double freq, total = 0;
    double result, p, log_result = 0;
    double entropy = 0;
    cout << "전체 빈도수 입력: ";
    cin >> total;
    while (1) {
        cout << "빈도수 입력: ";
        cin >> freq;
        if (freq == -1) {
            break;
        }
        p = freq / total;
        log_result = log2(1/p);
        result = p * log_result;
        entropy += result;
        cout << result << endl;
    }
    cout << endl << entropy << endl;

    return 0;
}
```

전체 빈도수와 각 빈도수를 입력하여 entropy를 구했다. 결과는 다음과 같다.

Microsoft Visual Studio 디버그 콘솔

```
빈도수 입력: 18  
0.023376  
빈도수 입력: 281  
0.194872  
빈도수 입력: 584  
0.310915  
빈도수 입력: 287  
0.197697  
빈도수 입력: 152  
0.12598  
빈도수 입력: 37  
0.0421795  
빈도수 입력: 9  
0.0130618  
빈도수 입력: 9  
0.0130618  
빈도수 입력: 9  
0.0130618  
빈도수 입력: 10  
0.0142811  
빈도수 입력: 80  
0.0776134  
빈도수 입력: -1  
  
4.44961
```

4.44961(bit)라는 결과가 나왔다. 앞서 결과에서 압축률이 45%정도 되는 것을 매우 entropy와 근접한 결과가 나왔다는 것을 확인할 수 있다.

본 과제를 통해 entropy coding이 얼마나 효율적인지 알게 되었다. 다만 파일의 크기가 작거나 ASCII가 골고루 나오는 경우 normal Huffman coding이 비교적 효율적이지 못할 것이라는 유추를 해볼 수 있는 과제였다. Huffman\_table의 크기는 거의 무시할 수 있을 정도로 작기 때문에 파일의 크기가 아주 작은 것이 아닌 이상 압축률은 이론상의 entropy와 매우 근접하게 나온다는 것을 확인했다.