HW1

2013312689 박찬열

Decrypting 방법 설명

Vingenere cipher를 해독하기 위해서 크게 두 단계, 1. key의 길이를 정하기. 2.각 자리의 key값을 구하기로 나누어서 문제를 해결하였다.

보다 원활하게 문제를 해결하기 위하여 여러 함수를 만들어 사용하였는데

1. void clear(int*prop) 함수는 문자열의 빈도수를 계산하기 위하여 만든 prop 배열을 초기화 하기 위해 만들었다. 안에는 0부터 255까지 탐색하는 for문이 딱 하나 있기 때문에 256번 탐색하며, 시간복잡도는 O(1)이라고 볼 수 있다.

2. double calc_prop(int* prop, int total, int len) 함수는 문자열 빈도수의 제곱을 구하기 위해 만들었다. 이 함수 역시 안에는 0부터 len (1바이트의 최대값 즉 255)까지 탐색하는 for문이 하나 있기 때문에 총 256번 탐색을 하며, 시간복잡도는 O(1)이라고 볼 수 있다.

3. int check(int* prop, int count, unsigned char now_key) 함수는 현재 key값(now_key)와 이 key값으로 decrypt하여 나온 문자의 빈도수 (prop)를 이용하여 now_key가 타당한 key 값인지 판단하는 함수이다. 이 함수에는 서로 독립된 for문이 총 두 개가 있지만 두 개 모두 알파벳의 개수만큼만 탐색하기 때문에 총 52번 탐색하며, 시간복잡도는 O(1)이라고 볼 수 있다.

이 세 함수 모두 main에서 돌아가는 for문에 비하여 그 회수가 적기 때문에 실질적으로 프로그램의 시간복잡도에는 크게 영향을 끼치지 못한다.

0. key length를 구하기 이전에

key length를 구하기 이전에 우선 txt파일을 열어 fseek함수와 ftell함수를 이용하여 문자열의 길이를 input_len 변수에 저장한 뒤, 암호화된 문자열을 input 배열에 input_len의 길이 만큼 읽어와 저장하였다.

1. key length 구하기

key length 를 구하기 위하여 우선 key length 의 값을 가정하고, 이것이 합당한지 반복하여 체크해주었다. 1 부터 10 까지 탐색하는 for 문, 즉 키의 길이를 추측하는 for 문 안에는 0 부터 시작하여 우리가 추측한 key length 길이만큼 탐색하는 for 문이 하나 더 있는데, 이는 한 자리에서만 key length 를 추측할 경우, key 가 '0x01, 0x02, 0x01, 0x03'인 경우 길이가 2 가 더 적합한 key length 로 계산되는 등 key length 값을 추측하는데 큰 오차가 발생하기 때문에 모든 자리에서 탐색 하기 위하여 존재한다.

key 의 length 를 찾기 위하여 input 배열에 저장된 암호문자열을 0 부터 key length 개의 자리부터 시작하여 key length 만큼 건너 뛰며 전체 문자열 길이를 넘지 않도록 input 배열의 문자를 읽어, 어떤 값을 가지고 있는지 확인 후 prop 배열에 0 부터 255 까지 어떤 문자열이 몇 번 나왔는지 기록한다. 이렇게 할 경우 input 에 있는 모든 문자를 탐색하게 되므로 어떤 key length L 에 대해서 총 input_len 번 탐색하며, 시간복잡도는 O(file_size)라고 할 수 있다.

그 뒤 이 prop 배열과 calc_prop 함수, 그리고 가장 적합한 key length 인 경우 모든 자리에서 문자의 분포가 plain 하지 않다는 점을 이용하여 calc_prop 로 구한 문자열의 빈도수 제곱의 평균을 구하고 모든 key length 에 대하여 가장 큰 평균 문자열 빈도수의 제곱을 가진 그때의 key length 를 Vingenere cipher 에 쓰인 key length 또는 key length 의 배수로 선택한다. 실제 key length 가 3 이며 key 값이 {0x01 0x02 0x03}인 경우 우리가 찾은 key length 는 3, 6, 9 는 평균 문자열의 빈도수 제곱은 거의 비슷하면서 약간씩 차이가 나는 값을 가지기 때문에 우리가 구한 key length 는 실제 key length 의 배수가 될 수 있다. 이 문제는 key 값을 구한 다음 해결하였다.
이 부분에서 calc_prop 는 총 10 번 호출되었으며, 파일의 내용을 전부 탐색하였기 때문에 총 10*256+file_size 번 탐색이 되었으며, 파일 크기는 1000 에서 5000 사이의 값을 가지므로 시간복잡도는 O(file_size)이다.

2. 각 자리의 key 값 구하기
각 자리의 key 값을 구하기 위하여 0 부터 255 까지 가능한 모든 key 값을 가정하며 input 파일을 decrypt 하여 이게 타당한 key 값인지 검사를 해주었다. 우선 i 번째 key 값을 candi (0 부터 255 까지의 수)라 가정한 다음 input 의 모든 i+key_length*n 자리를 탐색하며 candi 값으로 XOR 연산을 이용하여 decrypt 하며, prop 배열에 decrypt 된 값이 몇 번 나왔는지 기록한다.
아스키 코드 값의 범위를 벗어나는 값이 있다면 현재 candi 값은 절대 key 값이 될 수 없으므로 다른 값을 탐색한다. 만약 모든 값이 아스키 코드 값의 범위라면 check 함수를 불러 이게 타당한 key 값인지 판단한다.
check 함수에서는 파라미터로 넘어온 prop 배열을 이용하여 candi 값으로 decrypt 된 입력의 알파벳 소문자들의 개수와 그것의 빈도수를 구하게 되는데, 여기서 현재 input 으로 구한 빈도수와 실제 알파벳의 빈도수를 곱한 것들의 합을 구하여 실제 알파벳 빈도수 제곱의 합과 차이가 가장 적은 candi 값을 기록하도록 하였다. 즉, 모든 candi 값을 거치면서 실제 빈도수와 가장 가까운 단어가 key 값으로 선택 된다.
이 부분에서 이중 for 문과 그 for 문 안의 while 문이 있는데, 첫 for 문은 key 의 length 가지를 탐색하며 안에 있는 for 문은 0 부터 255 까지 즉 256 가지를 탐색한다, 또한 이 for 문 안에 while 문이 하나, 그리고 check 함수가 있는데 while 문은 최대 파일 크기 만큼 탐색하며, check 함수의 경우 총 52 번 탐색하기 때문에 이 부분은 최대 key_length*256*(file_size+52)번 탐색하며 이 것의 시간 복잡도는 O(key_length*file_size)라고 할 수 있다.

key 값을 구한 뒤
key 값을 구한 뒤 key 값이 반복되는 패턴인지 아닌지를 파악하여 가장 길이가 짧은 형태로 출력해준다. 반복하는 형태일 경우 실제 key length 는 현재 key length 의 약수만 가능하므로, 현재 key length 의 자기자신을 뺀 약수를 d 라고 했을 때, 모든 0 이상의 i+d<key_length 인 i 에 대하여 key[i] =key[i+d] 가 성립하면, key 는 d 마다 반복 되므로 key 의 길이를 d 라고 할 수 있다.
마지막으로 key 값과 input 을 key 로 decrypt 한 결과물을 hw1_output.txt 파일에 출력한다.

문제를 해결하기 위해 쓰인 이 코드의 시간복잡도는 가장 큰 시간복잡도를 가진 각 자리의 key 값 구하기 부분의 시간복잡도와 같으며, 이는 O(key_length * file_size)가 된다.

코드

```
#define _CRT_SECURE_NO_WARNINGS


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

#define MAX 5001                //파일 크기의 최대는 MAX(5000바이트)
#define Eng_l 0.0656010         //실제 영어 글자의 빈도수

double min_res;
unsigned char min_ind;
double original[26] = {
        0.08167,
        0.01492,
        0.02782,
        0.04253,
        0.12702,
        0.02228,
        0.02015,
        0.06094,
        0.06966,
        0.00153,
        0.00772,
        0.04025,
        0.02406,
        0.06749,
        0.07507,
        0.01929,
        0.00950,
        0.05987,
        0.06327,
        0.09056,
        0.02758,
        0.00978,
        0.02360,
        0.00160,
        0.01974,
        0.00074
};      //각 알파벳의 빈도수

int repeat[11][4] =
{ { 0 },{ 0 },{ 1,1 },{ 1,1 },{ 2,1,2 },{ 1,1 },{ 3,1,2,3 },{ 1,1 },{ 3,1,2,4 },{ 2,1,3 },{ 3,1
,2,5 } };
//반복되는 답을 제거하기 위한 배열
//n번째 배열의 첫칸에는 n의 약수의 개수 m이, 그 뒤로 m칸에는 1을 포함하고 자신을 포함하지 않는
약수가 들어있다.

void clear(int* prop)    //빈도수를 저장하는 prop배열의 초기화
{
        int i;
```

```
        for (i = 0; i < 256; i++)
                prop[i] = 0;
}

double calc_prop(int* prop, int total, int len)    //prop 배열에 저장된 빈도수의 제곱을 계산
{
        int i;
        double result = 0.0;

        for (i = 0; i < len; i++)
                result += ((double)prop[i] / (double)total)*((double)prop[i] / (double)total);

        return result;
}

int check(int* prop, int count, unsigned char now_key)     //현재 키값(now_key)값이 타당한지
확인하는 함수
{
        int i;
        int total = 0;
        double res = 0.0;

        for (i = 97; i < 123; i++)         //알파벳 소문자 개수를 센다
                total += prop[i];

        if (total == 0)  //알파벳 소문자 범위 내의 문자가 하나도 없으면
                return 0;

        //원래 알파벳의 빈도수와 key값으로 decrypt해서 얻은 빈도수를 곱한다
        for (i = 97; i < 123; i++)
                res += ((double)prop[i] / (double)total)*original[i - 'a'];



        if ((double)count*0.5>(double)total)
                return 0;


        //위에서 구한 값과 실제 알파벳 빈도수제곱이 거의 비슷하면 그것이 key의 값이다
        if (res > 0.06 && res < 0.8)
        {
                if (res < Eng_I)
                        res = Eng_I - res;
                else
                        res = res - Eng_I;
                if (res < 0.0)
                        res *= -1.0;
                if (res <= min_res)
                {
                        min_res = res;
                        min_ind = now_key;
                }
                return 1;
```

```
        }
        else
                return 0;

}

int main()
{
        int Tcase;       //키의 길이
        int i, j, k, candi, count;
        int input_len;   //파일의 길이
        int flag = 0;

        double max = -1.0;
        double tmp;
        double min = 1.0;
        int max_index;

        int prop[256] = { 0 };

        FILE* fIn;
        FILE* fOut;

        unsigned char ch;
        unsigned char tmp2;

        unsigned char* key;
        unsigned char input[MAX] = { 0 };

        fIn = fopen("hw1_input.txt", "r");
        fOut = fopen("hw1_output.txt", "wb");

        //파일의 길이 측정 및 파일의 값을 읽어와 input에 저장
        fseek(fIn, 0L, SEEK_END);
        input_len = ftell(fIn);
        fseek(fIn, 0L, SEEK_SET);
        fread(input, sizeof(unsigned char), input_len, fIn);


////////////////////////////////////////////////////////////// 코드블럭
1//////////////////////////////////////////////////////////////////////
        //key의 length를 결정
        for (Tcase = 1; Tcase <= 10; Tcase++)
        {
                k = 0;
                min = 0.0;
                for (k = 0; k < Tcase; k++)       //key의 길이를 구하기 위해 k부터 시작해서
key의 길이 번째 자리마다 글자를 수집
                {
                        i = 0;
                        j = 0;
                        clear(prop);
                        count = 0;
```

```
                        while (i + k < input_len)
                        {
                                prop[(int)input[i + k]]++;
                                i += Tcase;
                                count++;
                                j++;
                        }
                        tmp = calc_prop(prop, count, 256); //수집한 글자들의 빈도수 제곱을
구한다
                        min += tmp;
                }
                min = min / Tcase;        //글자 빈도수 제곱의 합의 평균을 구한다

                if (max < min)
                {
                        max = min;
                        max_index = Tcase;
                }
        }
        Tcase = max_index;        //글자들의 빈도수 제곱의 평균이 가장 큰 것을 key의 길이로
잡는다
        key = (unsigned char*)malloc(sizeof(unsigned char)*Tcase); //key값을 기록할 배열할당
////////////////////////////////////////////////////////////// 코드블럭
1//////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////// 코드블럭
2//////////////////////////////////////////////////////////////////////
        for (i = 0; i < Tcase; i++)       //key의 i번째 자리를 구함
        {
                max = -1.0;
                min_res = 1.0;
                min_ind = 0;
                for (candi = 0; candi < 256; candi++)     //key값의 후보들. 0~255까지 총
256가지
                {
                        key[i] = candi;
                        j = i;
                        clear(prop);
                        count = 0;
                        flag = 0;
                        while (j < input_len)
                        {
                                tmp2 = (key[i] ^ input[j]);

                                if ((tmp2 < 0) || (tmp2 > 127))   //input의 글자와 key값의
후보를 XOR 연산으로 계산하여 만약 아스키코드 범위를 벗어나면 후보에서 제외
                                {
                                        flag++;
                                        break;
                                }
```

```
                                else
                                {
                                        prop[(int)tmp2]++;
                                        count++;
                                }
                                j += Tcase;
                        }
                        if (flag)
                                continue;
                        else    //현재의 key값이 타당한지 check함수를 통해 확인
                                check(prop, count, key[i]);
                }
                key[i] = min_ind;
        }
////////////////////////////////////////////////////////////// 코드블럭
2//////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////// 코드블럭
3//////////////////////////////////////////////////////////////////////
        //key의 길이가 n일경우 n의 배수이면서 같은 패턴을 반복하는 경우가 구해질 수 있음
        //그럴경우 key의 값을 반복하는 패턴이 아닌 최소 단위로 변경해준다.
        for (i = 1; i <= repeat[Tcase][0]; i++)           //현재 key 길이의 약수의 개수 만큼
실행
        {
                flag = 0;
                for (j = 0; j + repeat[Tcase][i] <Tcase; j++)     //만약 모든 key[i]가
key[i+key길이의 약수]와 같으면 반복되는 패턴
                        if (key[j] != key[j + repeat[Tcase][i]])
                        {
                                flag = 1;
                                break;
                        }
                if (flag == 1)
                        continue;
                else
                        Tcase = repeat[Tcase][i];//반복되는 패턴이라면 key의 길이를 줄인다.
        }
////////////////////////////////////////////////////////////// 코드블럭
3//////////////////////////////////////////////////////////////////////


        for (i = 0; i < Tcase; i++)       //key값 출력
                fprintf(fOut, "0x%02x ", key[i]);
        fprintf(fOut, "₩n");

        for (i = 0; i<input_len; ++i)     //암호화 해제
        {
                input[i] ^= key[i % Tcase];
                ch = input[i];
                fwrite(&ch, sizeof(input[i]), 1, fOut);
        }
```

```
        free(key);

        fclose(fIn);
        fclose(fOut);


}
```