<Assignment 1>

Breaking a Vigenere variant cipher

소프트웨어학과
2019313065
박다영

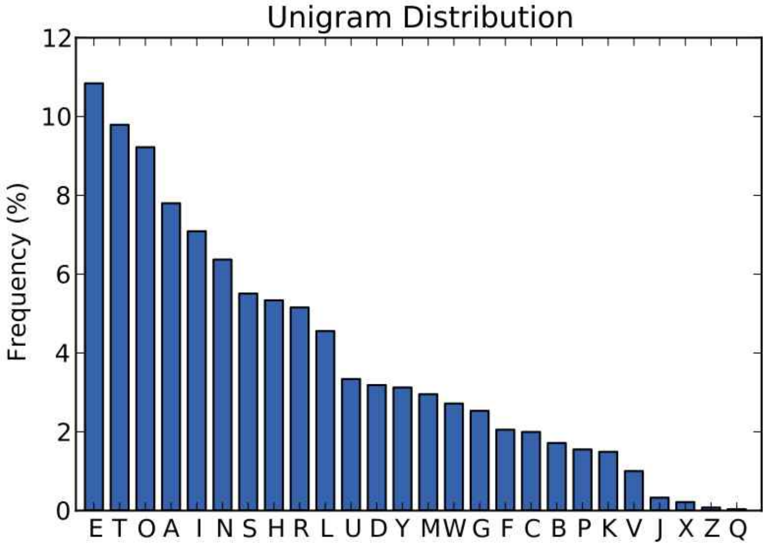
1. Instructions to illustrate how my code works

The goal of this assignment is to break vigenere variant where byte-wise XOR is used instead of addition modulo 26. There are three steps to break vigenere variant. First step is to find the length of key. Next, we should find the value of key. Finally, we should find plaintext using this key.

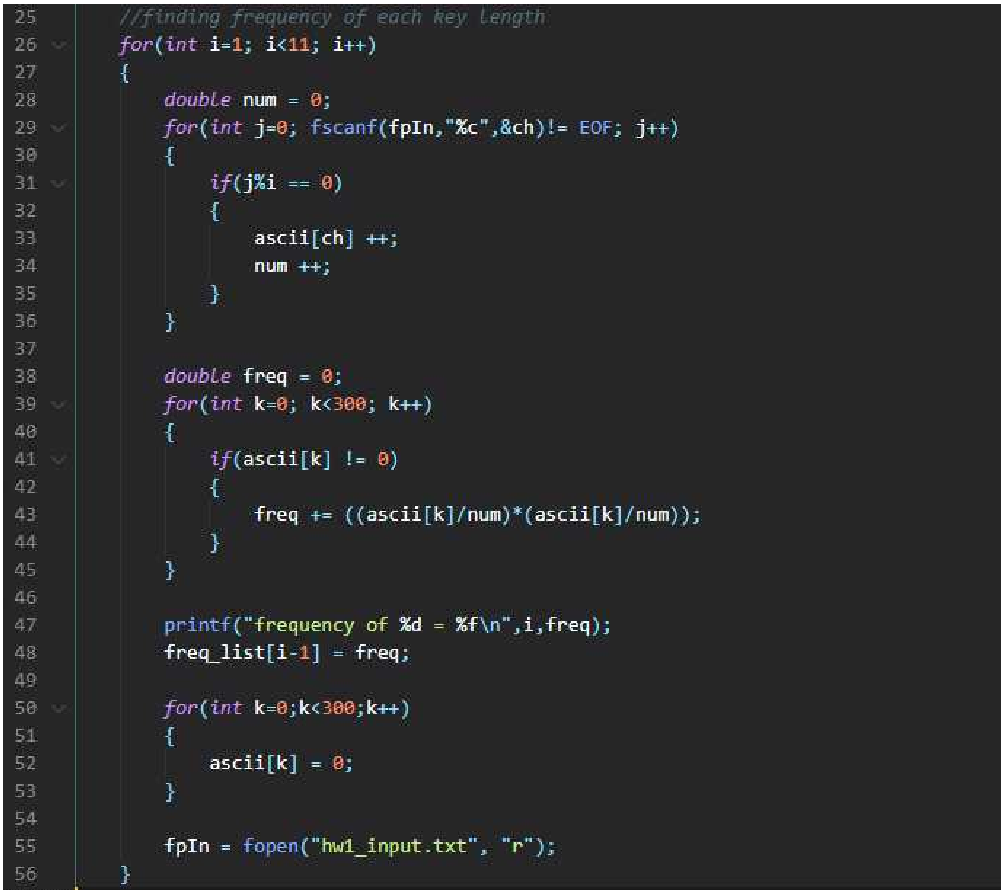
To find the length of key(key length is from 1 to 10), I calculated all frequencies in each case of length=1, 2, 3,,,10. Then, sort them in descending order. In most case, key length has the highest frequency, but there are some cases that multiple of that key length has the highest frequency. So if the difference of two highest frequencies are less than 0.01, I calculated the gdc of them(=save it in variable can), and find most biggest multiple of can which is smaller than 11, and find key value of it, and determine final key length.

Next, to find the value of key, I had to assume the most frequent letter in English, and find the most frequent letter in ciphertext and ^ the two letter. I considered 8 cases ; the most frequent letter is ‘ ’, ‘e’, ‘t’, ‘a’, ‘i’, ‘n’, ‘o’, ‘s’. Then, I calculated key value and changed ciphertext to plaintext for each case. I calculated the frequency of lowercase letters and ‘ ’ for each case and take maximum value.

Finally, I wrote the value of key(change key value to hex) and plaintext to output file.

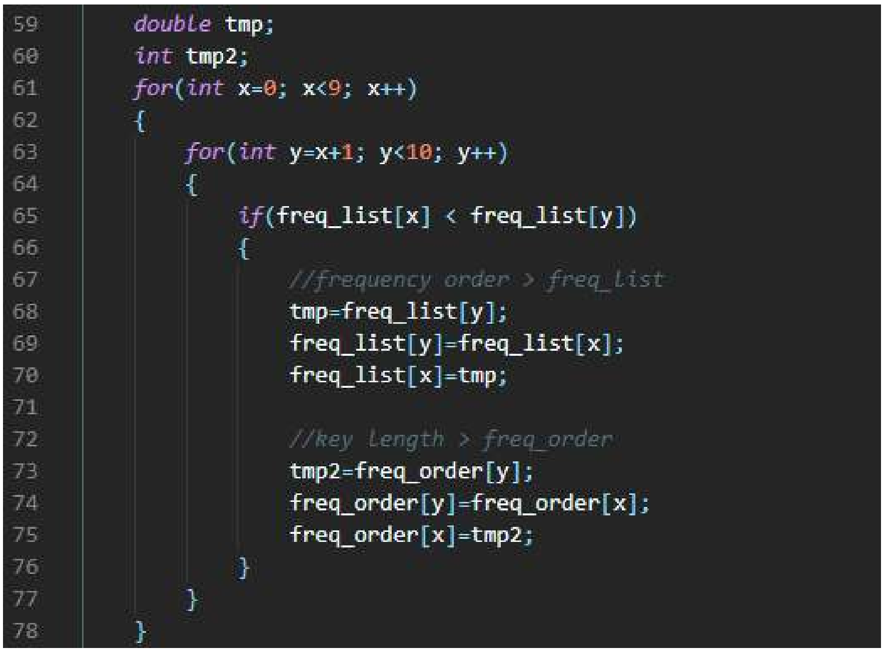


2. Explanation of my code



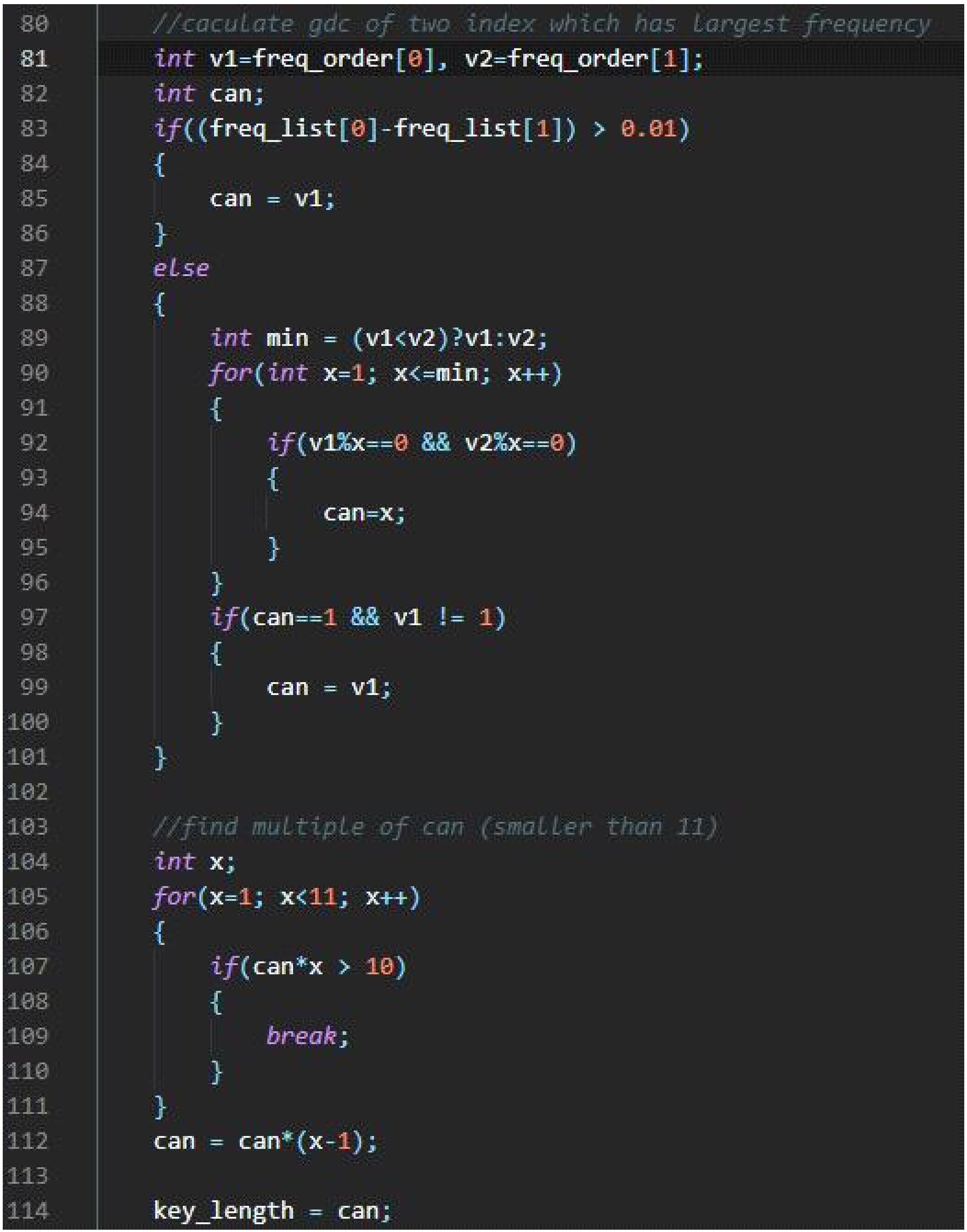
```c
    //finding frequency of each key length
    for(int i=1; i<11; i++)
    {
        double num = 0;
        for(int j=0; fscanf(fpIn,"%c",&ch)!= EOF; j++)
        {
            if(j%i == 0)
            {
                ascii[ch] ++;
                num ++;
            }
        }

        double freq = 0;
        for(int k=0; k<300; k++)
        {
            if(ascii[k] != 0)
            {
                freq += ((ascii[k]/num)*(ascii[k]/num));
            }
        }

        printf("frequency of %d = %f\n",i,freq);
        freq_list[i-1] = freq;

        for(int k=0;k<300;k++)
        {
            ascii[k] = 0;
        }

        fpIn = fopen("hw1_input.txt", "r");
    }
```

I calculated frequencies of letters in ciphertext for every nth character(n = 1,2, 3,,,10). Then saved frequency in freq_list array.



```c
    double tmp;
    int tmp2;
    for(int x=0; x<9; x++)
    {
        for(int y=x+1; y<10; y++)
        {
            if(freq_list[x] < freq_list[y])
            {
                //frequency order > freq_list
                tmp=freq_list[y];
                freq_list[y]=freq_list[x];
                freq_list[x]=tmp;

                //key length > freq_order
                tmp2=freq_order[y];
                freq_order[y]=freq_order[x];
                freq_order[x]=tmp2;
            }
        }
    }
```

I sorted frequency in decending order. I saved value of frequency in freq_list, and corresponding key length in freq_order.

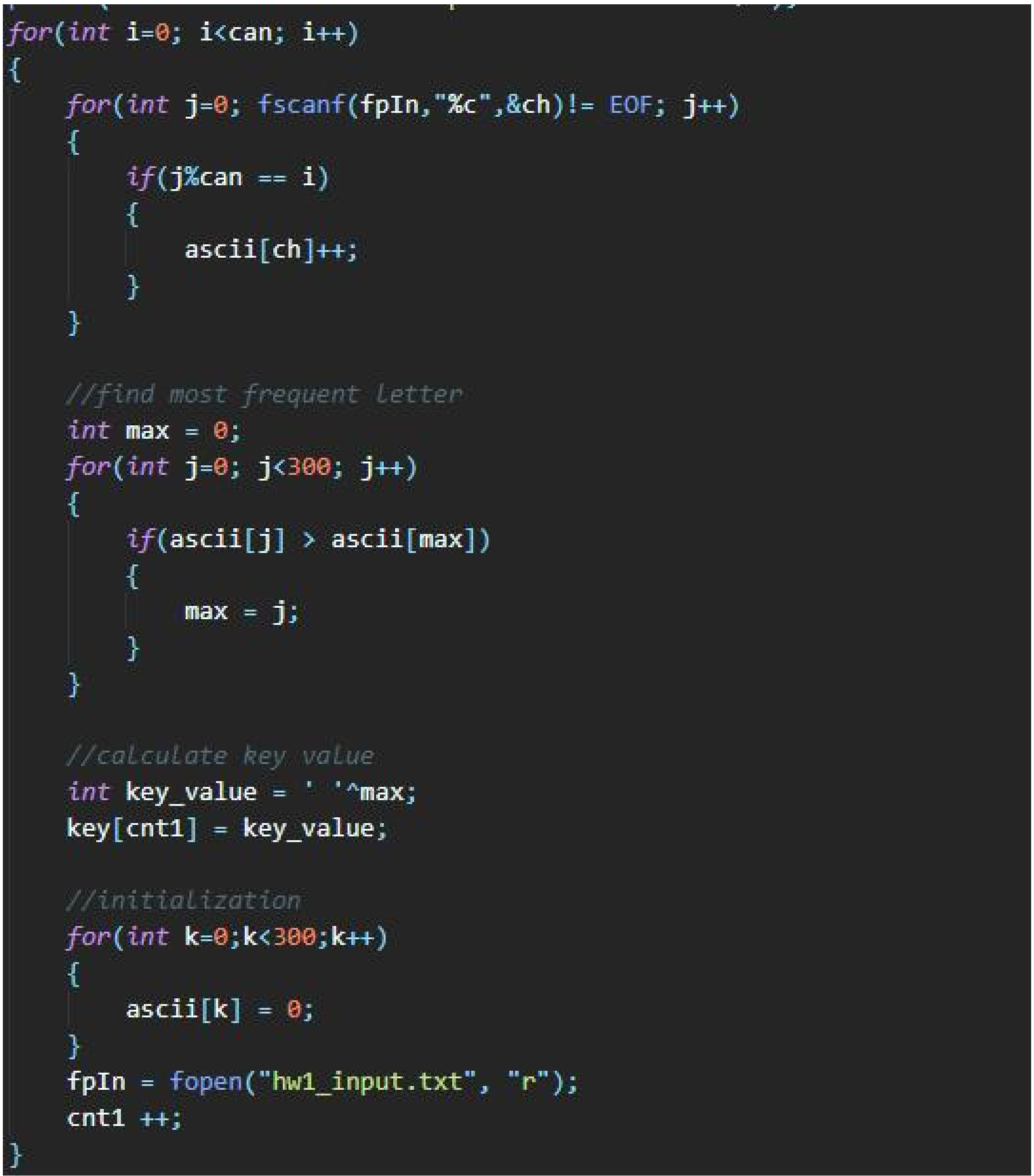


```
    //caculate gdc of two index which has largest frequency
    int v1=freq_order[0], v2=freq_order[1];
    int can;
    if((freq_list[0]-freq_list[1]) > 0.01)
    {
        can = v1;
    }
    else
    {
        int min = (v1<v2)?v1:v2;
        for(int x=1; x<=min; x++)
        {
            if(v1%x==0 && v2%x==0)
            {
                can=x;
            }
        }
        if(can==1 && v1 != 1)
        {
            can = v1;
        }
    }

    //find multiple of can (smaller than 11)
    int x;
    for(x=1; x<11; x++)
    {
        if(can*x > 10)
        {
            break;
        }
    }
    can = can*(x-1);

    key_length = can;
```

v1 and v2 are value of key lengths which has most highest frequency. Subtract two and if it is less than 0.01, calculate gdc of two and save it in variable can. Find biggest multiple of can which is less than 11 and save it in key_length.

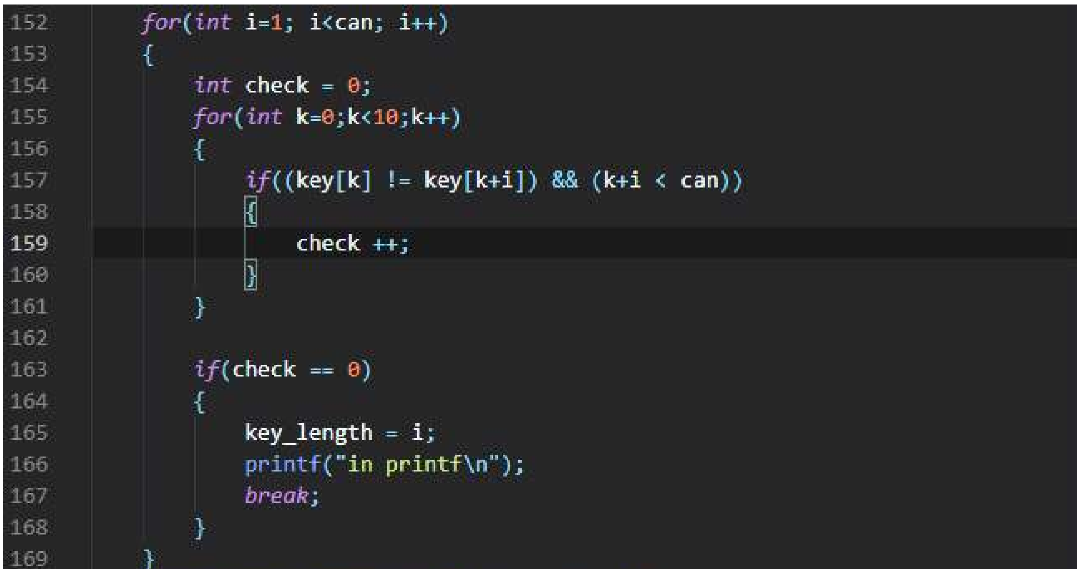
Next, we should find key value.

First, I assumed most frequent letter in English is ‘ ’.



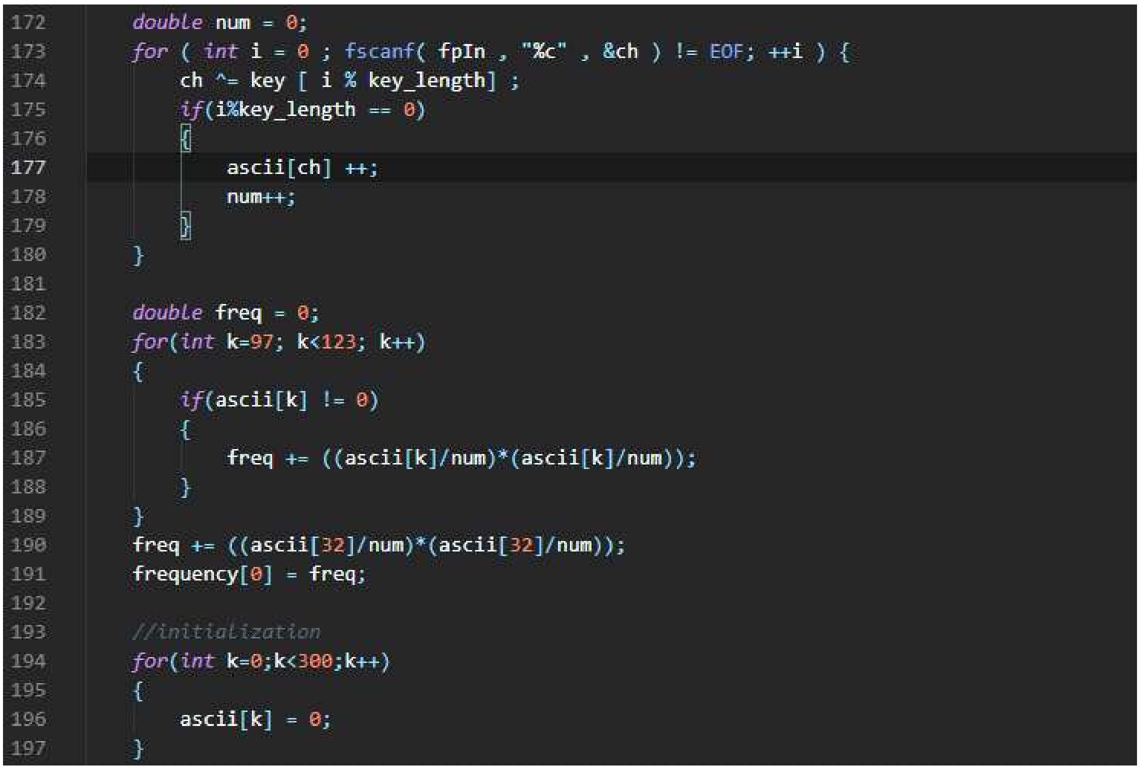
```
for(int i=0; i<can; i++)
{
    for(int j=0; fscanf(fpIn,"%c",&ch)!= EOF; j++)
    {
        if(j%can == i)
        {
            ascii[ch]++;
        }
    }

    //find most frequent letter
    int max = 0;
    for(int j=0; j<300; j++)
    {
        if(ascii[j] > ascii[max])
        {
            max = j;
        }
    }

    //calculate key value
    int key_value = ' '^max;
    key[cnt1] = key_value;

    //initialization
    for(int k=0;k<300;k++)
    {
        ascii[k] = 0;
    }
    fpIn = fopen("hw1_input.txt", "r");
    cnt1 ++;
}
```

Find most frequent letter in ciphertext and save it to max. Calculate ‘ ’^max and it is the value of key.



```
for(int i=1; i<can; i++)
{
    int check = 0;
    for(int k=0;k<10;k++)
    {
        if((key[k] != key[k+i]) && (k+i < can))
        {
            check ++;
        }
    }

    if(check == 0)
    {
        key_length = i;
        printf("in printf\n");
        break;
    }
}
```

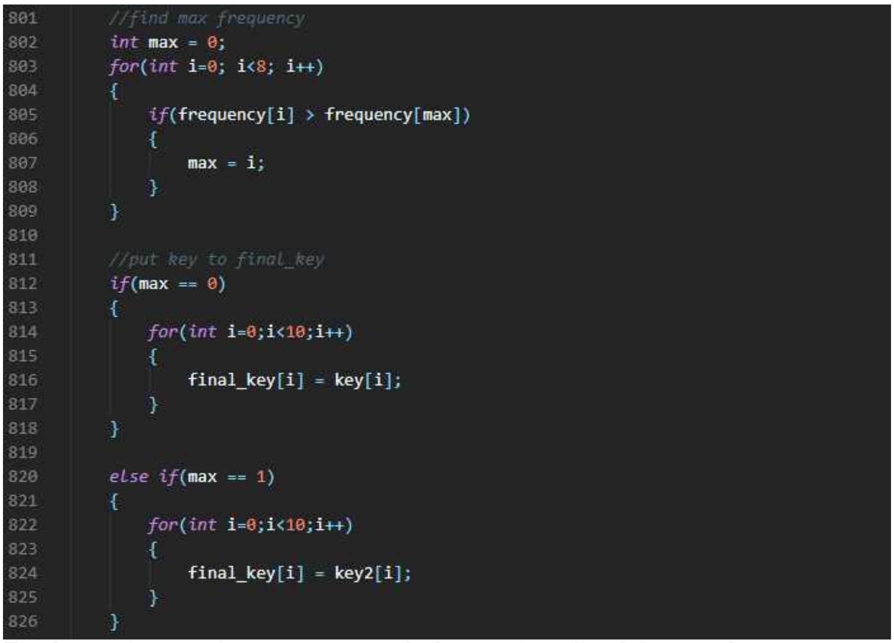
If key value is repeated, find the smallest length representing value of key and save it. For example, if key value is 0x12 0x23 0x12 0x23 .... save key length as 2. We found the final key length.



```
double num = 0;
for ( int i = 0 ; fscanf( fpIn , "%c" , &ch ) != EOF; ++i ) {
    ch ^= key [ i % key_length] ;
    if(i%key_length == 0)
    {
        ascii[ch] ++;
        num++;
    }
}

double freq = 0;
for(int k=97; k<123; k++)
{
    if(ascii[k] != 0)
    {
        freq += ((ascii[k]/num)*(ascii[k]/num));
    }
}
freq += ((ascii[32]/num)*(ascii[32]/num));
frequency[0] = freq;

//initialization
for(int k=0;k<300;k++)
{
    ascii[k] = 0;
}
```

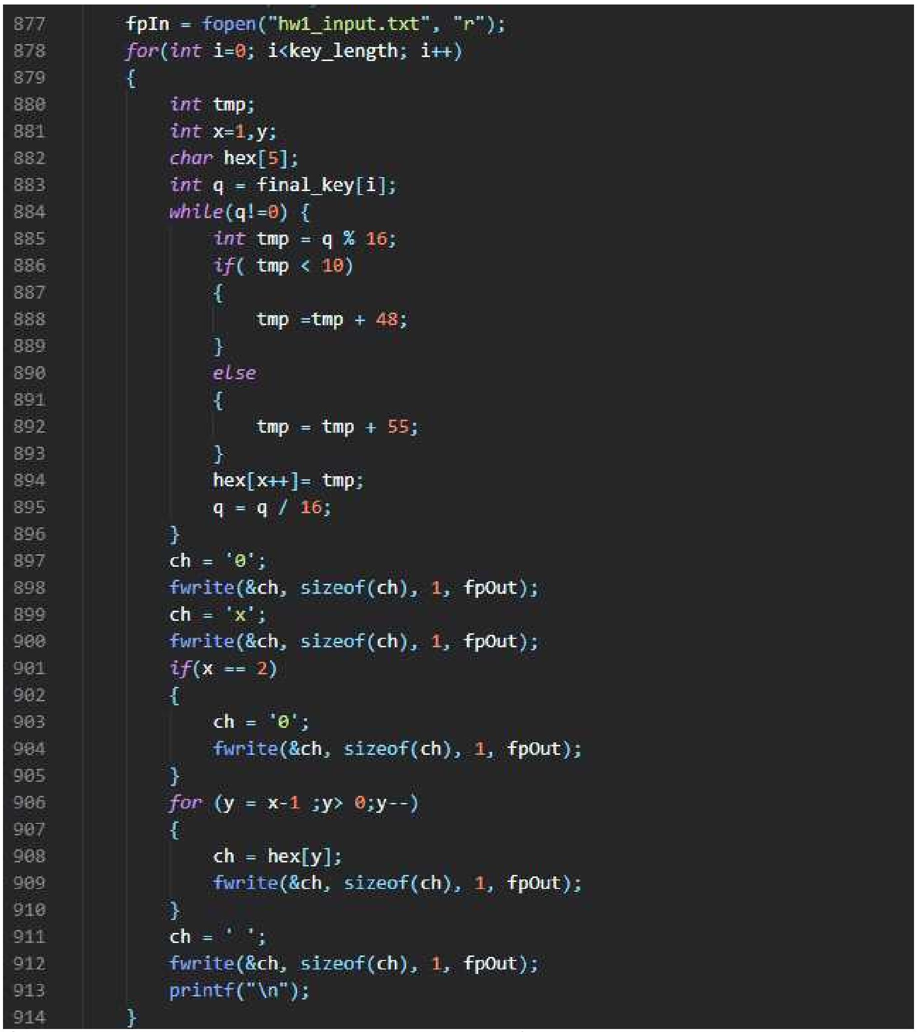
Change ciphertext into plaintext. Calculate letters in plaintext, and calculate frequencies of lowercase letters and ‘ ’. Save frequency in frequency array.

Repeat these steps while assuming most frequent letter in English is ‘e’, ‘t’, ‘a’, ‘i’, ‘n’, ‘o’, ‘s’.



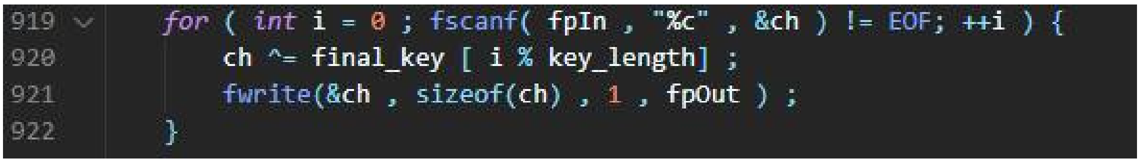
```c
    //find max frequency
    int max = 0;
    for(int i=0; i<8; i++)
    {
        if(frequency[i] > frequency[max])
        {
            max = i;
        }
    }

    //put key to final_key
    if(max == 0)
    {
        for(int i=0;i<10;i++)
        {
            final_key[i] = key[i];
        }
    }

    else if(max == 1)
    {
        for(int i=0;i<10;i++)
        {
            final_key[i] = key2[i];
        }
    }
```

Find maximum frequency, and determine final key value.



```c
    fpIn = fopen("hw1_input.txt", "r");
    for(int i=0; i<key_length; i++)
    {
        int tmp;
        int x=1,y;
        char hex[5];
        int q = final_key[i];
        while(q!=0) {
            int tmp = q % 16;
            if( tmp < 10)
            {
                tmp =tmp + 48;
            }
            else
            {
                tmp = tmp + 55;
            }
            hex[x++]= tmp;
            q = q / 16;
        }
        ch = '0';
        fwrite(&ch, sizeof(ch), 1, fpOut);
        ch = 'x';
        fwrite(&ch, sizeof(ch), 1, fpOut);
        if(x == 2)
        {
            ch = '0';
            fwrite(&ch, sizeof(ch), 1, fpOut);
        }
        for (y = x-1 ;y> 0;y--)
        {
            ch = hex[y];
            fwrite(&ch, sizeof(ch), 1, fpOut);
        }
        ch = ' ';
        fwrite(&ch, sizeof(ch), 1, fpOut);
        printf("\n");
    }
```

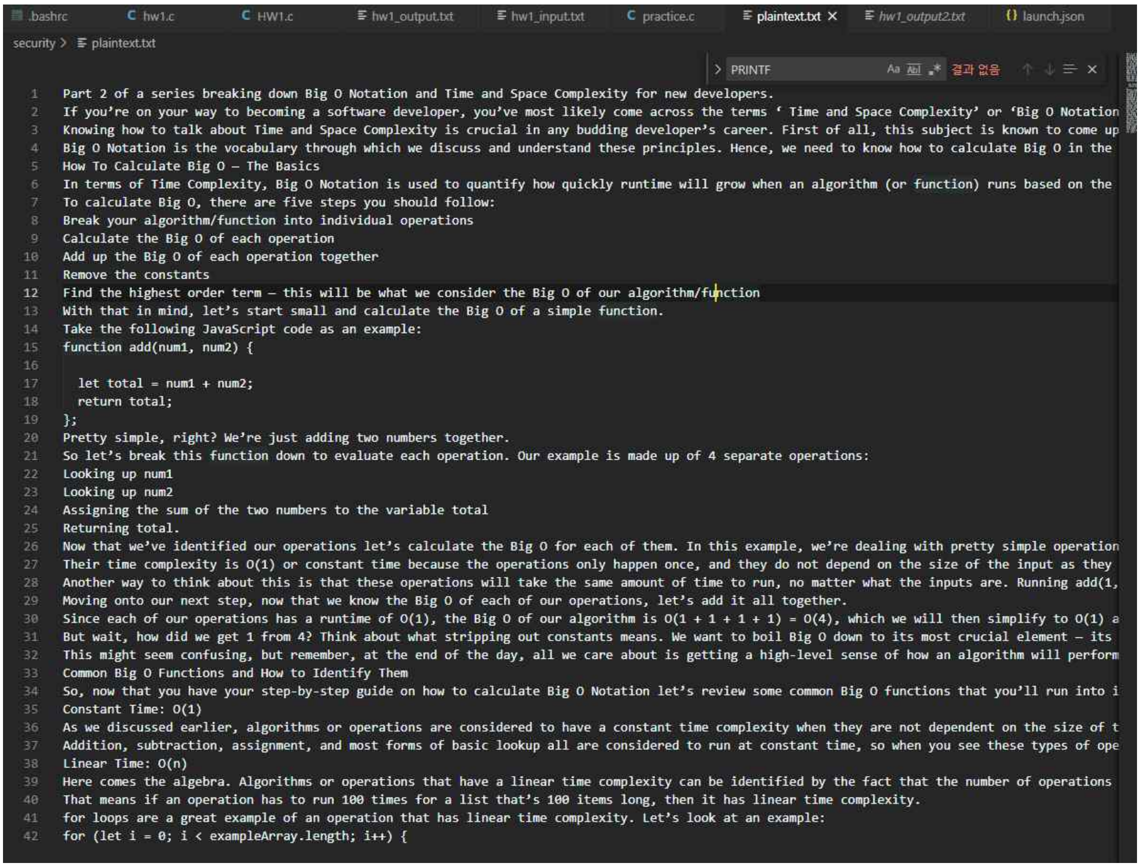
Change to key value to hex, and put 0x ahead of it, and save it in output file.



```
for ( int i = 0 ; fscanf( fpIn , "%c" , &ch ) != EOF; ++i ) {
    ch ^= final_key [ i % key_length] ;
    fwrite(&ch , sizeof(ch) , 1 , fpOut ) ;
}
```

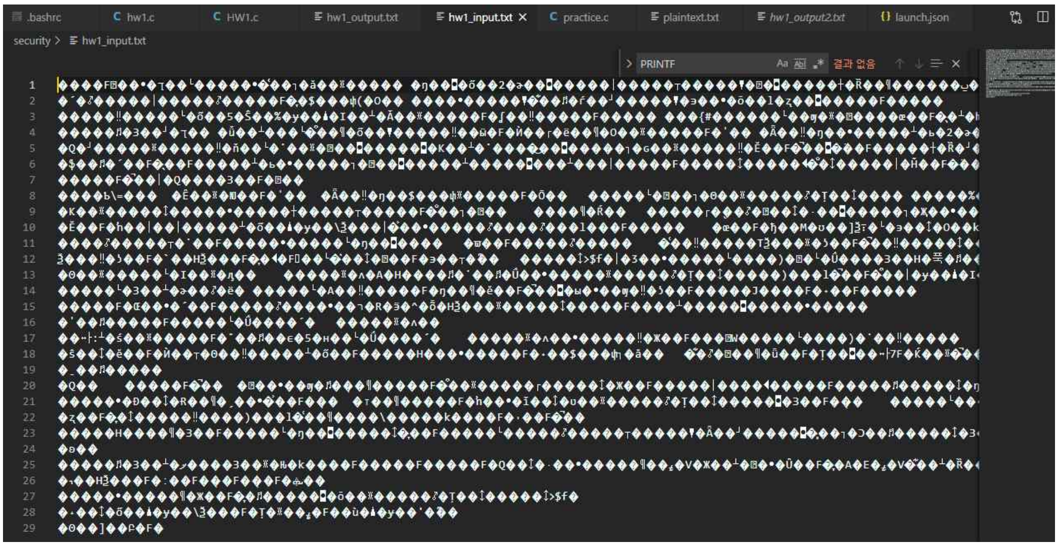
Change ciphertext to plaintext and save it in ouput file.

I'll show you the example.

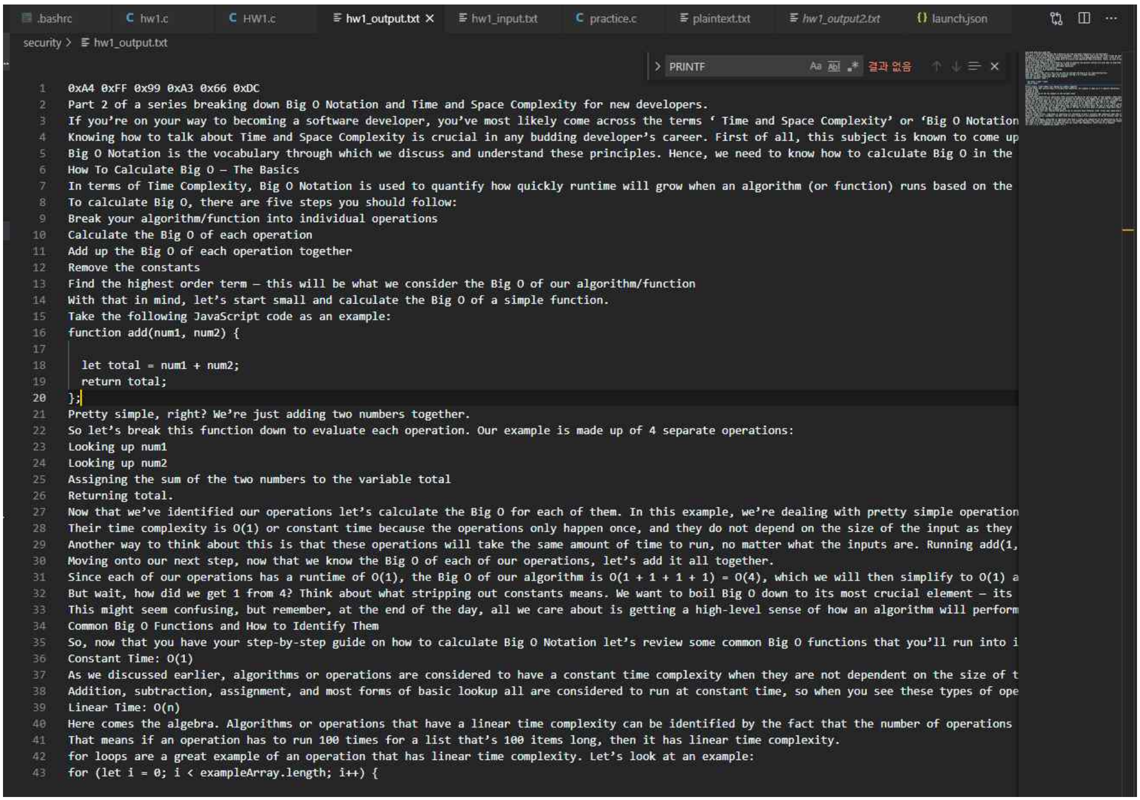


```
Part 2 of a series breaking down Big O Notation and Time and Space Complexity for new developers.
If you're on your way to becoming a software developer, you've most likely come across the terms ' Time and Space Complexity' or 'Big O Notation
Knowing how to talk about Time and Space Complexity is crucial in any budding developer's career. First of all, this subject is known to come up
Big O Notation is the vocabulary through which we discuss and understand these principles. Hence, we need to know how to calculate Big O in the
How To Calculate Big O – The Basics
In terms of Time Complexity, Big O Notation is used to quantify how quickly runtime will grow when an algorithm (or function) runs based on the
To calculate Big O, there are five steps you should follow:
Break your algorithm/function into individual operations
Calculate the Big O of each operation
Add up the Big O of each operation together
Remove the constants
Find the highest order term – this will be what we consider the Big O of our algorithm/function
With that in mind, let's start small and calculate the Big O of a simple function.
Take the following JavaScript code as an example:
function add(num1, num2) {

  let total = num1 + num2;
  return total;
};
Pretty simple, right? We're just adding two numbers together.
So let's break this function down to evaluate each operation. Our example is made up of 4 separate operations:
Looking up num1
Looking up num2
Assigning the sum of the two numbers to the variable total
Returning total.
Now that we've identified our operations let's calculate the Big O for each of them. In this example, we're dealing with pretty simple operation
Their time complexity is O(1) or constant time because the operations only happen once, and they do not depend on the size of the input as they
Another way to think about this is that these operations will take the same amount of time to run, no matter what the inputs are. Running add(1,
Moving onto our next step, now that we know the Big O of each of our operations, let's add it all together.
Since each of our operations has a runtime of O(1), the Big O of our algorithm is O(1 + 1 + 1 + 1) = O(4), which we will then simplify to O(1) a
But wait, how did we get 1 from 4? Think about what stripping out constants means. We want to boil Big O down to its most crucial element – its
This might seem confusing, but remember, at the end of the day, all we care about is getting a high-level sense of how an algorithm will perform
Common Big O Functions and How to Identify Them
So, now that you have your step-by-step guide on how to calculate Big O Notation let's review some common Big O functions that you'll run into i
Constant Time: O(1)
As we discussed earlier, algorithms or operations are considered to have a constant time complexity when they are not dependent on the size of t
Addition, subtraction, assignment, and most forms of basic lookup all are considered to run at constant time, so when you see these types of ope
Linear Time: O(n)
Here comes the algebra. Algorithms or operations that have a linear time complexity can be identified by the fact that the number of operations
That means if an operation has to run 100 times for a list that's 100 items long, then it has linear time complexity.
for loops are a great example of an operation that has linear time complexity. Let's look at an example:
for (let i = 0; i < exampleArray.length; i++) {
```

This is the plaintext in plaintext.txt



This is the ciphertext / key value is 0xA4 0xFF 0x99 0x66 0xDC



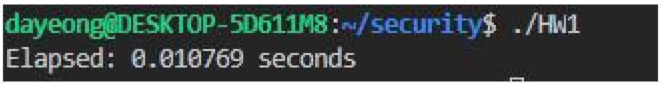
```
0xA4 0xFF 0x99 0xA3 0x66 0xDC
Part 2 of a series breaking down Big O Notation and Time and Space Complexity for new developers.
If you're on your way to becoming a software developer, you've most likely come across the terms ' Time and Space Complexity' or 'Big O Notation
Knowing how to talk about Time and Space Complexity is crucial in any budding developer's career. First of all, this subject is known to come up
Big O Notation is the vocabulary through which we discuss and understand these principles. Hence, we need to know how to calculate Big O in the
How To Calculate Big O – The Basics
In terms of Time Complexity, Big O Notation is used to quantify how quickly runtime will grow when an algorithm (or function) runs based on the
To calculate Big O, there are five steps you should follow:
Break your algorithm/function into individual operations
Calculate the Big O of each operation
Add up the Big O of each operation together
Remove the constants
Find the highest order term – this will be what we consider the Big O of our algorithm/function
With that in mind, let's start small and calculate the Big O of a simple function.
Take the following JavaScript code as an example:
function add(num1, num2) {

  let total = num1 + num2;
  return total;
};
Pretty simple, right? We're just adding two numbers together.
So let's break this function down to evaluate each operation. Our example is made up of 4 separate operations:
Looking up num1
Looking up num2
Assigning the sum of the two numbers to the variable total
Returning total.
Now that we've identified our operations let's calculate the Big O for each of them. In this example, we're dealing with pretty simple operation
Their time complexity is O(1) or constant time because the operations only happen once, and they do not depend on the size of the input as they
Another way to think about this is that these operations will take the same amount of time to run, no matter what the inputs are. Running add(1,
Moving onto our next step, now that we know the Big O of each of our operations, let's add it all together.
Since each of our operations has a runtime of O(1), the Big O of our algorithm is O(1 + 1 + 1 + 1) = O(4), which we will then simplify to O(1) a
But wait, how did we get 1 from 4? Think about what stripping out constants means. We want to boil Big O down to its most crucial element – its
This might seem confusing, but remember, at the end of the day, all we care about is getting a high-level sense of how an algorithm will perform
Common Big O Functions and How to Identify Them
So, now that you have your step-by-step guide on how to calculate Big O Notation let's review some common Big O functions that you'll run into i
Constant Time: O(1)
As we discussed earlier, algorithms or operations are considered to have a constant time complexity when they are not dependent on the size of t
Addition, subtraction, assignment, and most forms of basic lookup all are considered to run at constant time, so when you see these types of ope
Linear Time: O(n)
Here comes the algebra. Algorithms or operations that have a linear time complexity can be identified by the fact that the number of operations
That means if an operation has to run 100 times for a list that's 100 items long, then it has linear time complexity.
for loops are a great example of an operation that has linear time complexity. Let's look at an example:
for (let i = 0; i < exampleArray.length; i++) {
```

This is the output.txt which includes value of key and plaintext.

3. Performance analysis

The time complexity of my code is O(n^2).

I calcuated runtime of my code with using clock function.



```
dayeong@DESKTOP-5D611M8:~/security$ ./HW1
Elapsed: 0.010769 seconds
```