现代密码学实验 - 1

1. 维吉尼亚无密钥破译

1.1 实验原理

维吉尼亚密码是一种多表代换密码，利用多个字母表的轮换来加密明文。无密钥破译的关键步骤包括猜测密钥长度、分组频率分析和最终的密钥确定。在实际破译中，常使用卡方分布分析每个密钥可能的移位值

具体步骤为：

- 使用 **Kasiski 测试法**找到所有可能的周期
- 通过**重合指数法**，基于卡方分布分析各个子密钥对应字母的频率分布，找到最有可能的密钥
- 将子密钥组合，解出原始的明文

卡方分布的核心是比较不同字符出现的实际频率与期望频率，通过最小化差异，确定各个子密钥的具体值

1.2 代码分析

`encode`： 存储从标准输入读取的密文。 `freq[]`： 用来保存当前处理的文本子集中的字符频率分布。 `decryptedScore`： 用于记录当前尝试的密钥下，密文的解密质量，越低的分数表示破译的质量越好

```
string encode;
float freq[26] = {};
float decryptedScore = 0.0;
```

将密文按给定的密钥长度进行分组，每组对应于密钥的一个字符。例如，假设密钥长度为 3，则第 1、4、7...个字符属于第 1 组，第 2、5、8...个字符属于第 2 组，以此类推

`encode`： 整个密文字符串。 `length`： 假设的密钥长度。返回值 `res` 是一个二维字符向量，每个子向量存储属于密钥相应位置的字符

```cpp
vector<vector<char>> splitText(string& encode, int length) {
    vector<vector<char>> res(length);
    int idx = 0;
    for (char ch : encode) {
        if (isalpha(ch)) {
            res[idx % length].push_back(ch);
            idx++;
        }
    }
    return res;
}
```

统计文本中每个字母的出现次数，并保存在 `freq[]` 中。然后将每个字母的计数除以字符总数，得到相应字母的频率

```cpp
void calFreq(vector<char>& text) {
    float tot = text.size();
    for (char ch : text) {
        if (isalpha(ch)) {
            freq[toupper(ch) - 'A'] += 1.0;
        }
    }
    for (int i = 0; i < 26; i++) {
        freq[i] /= tot;
    }
    return;
}
```

通过对每个可能的移位进行卡方分布计算，找到密钥的最优移位量

对每个可能的移位 $shift$(从 0 到 25)，计算密文中字符的移位结果。调用 `calFreq()` 函数计算移位后文本的字母频率分布。使用卡方检验(`chiSquare`)比较实际频率与标准英语字母频率(`alphaRate[]`)，找到使卡方值最小的移位量 `bestShift` 。将该移位量返回，表示该子密文的最佳密钥字符

```
int findBestShift(vector<char>& text) {
    float minChiSquare = 1e9;
    int bestShift = 0;

    for (int shift = 0; shift < 26; shift++) {
        vector<char> shiftedText = text;
        for (char &ch : shiftedText) {
            ch = (toupper(ch) - 'A' - shift + 26) % 26 + 'A';
        }

        calFreq(shiftedText);
        float chiSquare = 0.0;
        for (int i = 0; i < 26; i++) {
            float temp = freq[i] - alphaRate[i];
            chiSquare += (temp * temp) / alphaRate[i];
        }

        if (chiSquare < minChiSquare) {
            minChiSquare = chiSquare;
            bestShift = shift;
        }
    }

    decryptedScore += minChiSquare;

    return bestShift;
}
```

根据生成的密钥对密文进行维吉尼亚解密

遍历密文，对于每个字母，使用密钥中对应位置的字母进行解密运算。对每个字符，根据其大小写情况，将其转换为对应的解密字符。输出解密后的文本

```
void decryptVigenere(string& ciphertext, string& key) {
    int keyLen = key.length();

    for (int i = 0, j = 0; i < (int)ciphertext.length(); i++) {
        if (isalpha(ciphertext[i])) {
            char base = isupper(ciphertext[i]) ? 'A' : 'a';
            ciphertext[i] = (ciphertext[i] - base - (toupper(key[j % keyLen]) - 'A') + 26) % 2
            j++;
        }
        printf("%c", ciphertext[i]);
    }

    return;
}
```

程序的主控制逻辑，逐步猜测密钥长度并进行破译，输出最终解密结果。

从输入中读取加密文本，将其存储在 `encode` 中。设置初始最优密钥长度范围（`key_max_length`），并开始从 4 到 `key_max_length` 尝试不同的密钥长度

对于每个密钥长度，使用 `splitText()` 将密文按密钥长度分组，逐组调用 `findBestShift()` 找到最优移位，并构造密钥

对每次构造出的密钥，计算其卡方分数，保留得分最低的密钥作为最终结果

使用最佳密钥调用 `decryptVigenere()` 解密密文，并输出解密结果

1.3 实现优化

在优化维吉尼亚密码的破解时，我们主要关注如何**更高效地**推测密钥的长度。在之前的代码中，我们使用了**暴力枚举法**来尝试所有可能的密钥长度。而在实际的破译过程中，**Kasiski 测试法**为我们提供了一种更优化的方式来猜测密钥长度。这里，我们仅考虑长度为 3 的相同密文段，并计算它们的距离来推测可能的周期

通过检测密文中长度为 3 的相同片段，利用它们之间的距离及其所有因数来确定可能的周期，可以减少我们在破译过程中的计算复杂度

将 Kasiski 测试法引入代码中，重点关注长度为 3 的重复密文段。通过计算这些片段的距离，并提取它们的所有因数(除 1 外)，我们可以得到可能的周期集合

```cpp
#include <set>

set<int> kasiski_test(string& encode) {
    string encode_str(encode.begin(), encode.end());
    set<int> candidates;
    int encode_length = encode.size();

    for (int i = 0; i < encode_length - 2; ++i) {
        string substr = encode_str.substr(i, 3);
        size_t found = encode_str.substr(i + 2, encode_length - i).find(substr);

        if (found != string::npos) {
            int distance = abs(i - (int)found);
            candidates.insert(distance);
        }
    }

    return candidates;
}
```

2. 仿射希尔密码分析

2.1 实验原理

仿射密码是一种经典的代换密码，它结合了线性代换和加法代换。希尔密码是基于矩阵乘法的一种多表代换密码，其加密过程利用矩阵对字母进行线性变换。密钥矩阵需要是可逆的，以便解密时进行逆运算

对于破译仿射希尔密码，核心步骤为：

- 找到密钥矩阵的逆矩阵，并利用它对加密的文本进行解密
- 如果没有密钥，可以通过已知的明文-密文对推导出密钥矩阵，进而进行解密

2.2 代码分析

创建两个矩阵 `X` 和 `Y`， `X`： 存储明文的 `m x m` 矩阵，每个字符转换为相应的数字（从 0 到 25）。 `Y`： 存储密文的 `m x m` 矩阵，使用同样的转换

```
matrix X(m, vector<int>(m));
matrix Y(m, vector<int>(m));

for (int i = 0; i < m; ++i) {
    for (int j = 0; j < m; ++j) {
        X[i][j] = decode[i * m + j] - 'A';
        Y[i][j] = encode[i * m + j] - 'A';
    }
}
```

`n`：计算出 `encode` 中包含多少个子块，每个子块的大小是 `m`。初始化 `A` 和 `B` 矩阵，以存储从明文和密文中获得的线性方程组

```
n = encode.size() / m;
matrix A = matrix(n, vector<int>(m));
matrix B = matrix(n, vector<int>(m));
matrix inv_A = matrix(m, vector<int>(n));
```

`ATA = matmul(A, A);`：计算矩阵 `A` 的转置与 `A` 的乘积，存储到 `ATA`。`det_inv = inverseTable[cal_det(ATA)];`：计算 `ATA` 的行列式，并查找其在 `inverseTable` 中的逆元。如果行列式为 0，则 `A` 不可逆，需要重新选择行。如果找到可逆的 `A`，就跳出循环，准备计算密钥

```
ATA = matmul(A, A);
det_inv = inverseTable[cal_det(ATA)];
if (det_inv != 0) break;
```

使用之前计算的逆矩阵 `cal_inv(ATA, det_inv)` 来计算密钥矩阵 `C`。首先计算 `ATA` 的逆矩阵。然后将 `A` 和 `B` 相乘，最后与逆矩阵相乘，得到密钥矩阵 `C`

计算 `X` 和 `C` 的乘积，然后将其从 `Y` 中相减，得到 `D`。这一步实际上是通过得到的密钥矩阵 `C`，将明文重新映射到原始密文空间

```
matrix C = matmul(cal_inv(ATA, det_inv), matmul(A, B));
matrix D = matsub(Y, matmul(X, C));
```

整个破译过程利用线性代数的原理，通过随机选择子块构造线性方程，进而求解出加密过程中使用的密钥。使用矩阵乘法、行列式计算、矩阵逆运算等数学工具，最终实现对仿射希尔密码的破解

2.3 实现优化

在仿射希尔密码的破译过程中，我们可能会遇到**非可逆矩阵**的问题，随机选择子块并构造线性方程的这一部分代码可以进行一些优化

```
while (true) {
    for (int i = 0; i < n; ++i) {
        int idx = dist(gen);
        for (int j = 0; j < m; ++j) {
            A[i][j] = (decode[i * m + j] - decode[idx * m + j] + 26) % 26;
            B[i][j] = (encode[i * m + j] - encode[idx * m + j] + 26) % 26;
            inv_A[j][i] = A[i][j];
        }
    }
}
```

使用随机数生成器生成一个随机的行索引 `idx` 。对于每个子块 `i` ，通过从 `decode` 和 `encode` 中相应字符的差值填充 `A` 和 `B` 矩阵。 `A[i][j]` 和 `B[i][j]` 代表子块 `i` 的第 `j` 列元素和随机选中的子块 `idx` 的对应列元素之差，这样可以保证行列式的计算会尽量避免为 0

这相当于构造一组线性方程，便于后续的求解

以上优化保证生成的线性方程组具有**较高的可逆性**。通过适当调整随机性和选择策略，可以在保持安全性的同时提高解密过程的效率