Universidade de Brasília
Departamento de Ciência da Computação
CIC0201 - Segurança Computacional, Turma 02
Professora: Lorena Borges

Lista de Exercícios 01

Iasmim de Queiroz Freitas
190108665

2025/01

Contents

1 Quebrando Shift Cipher

Nessa primeira parte da lista, foi testada a cifra por deslocamento, que consiste em deslocar as letras do alfabeto um certo número de posições. Essa cifra pode ser feita utilizando uma chave, onde, para cada posição da cifra, ocorre o deslocamento correspondente ao valor da chave no índice atual (módulo o tamanho da chave). A primeira cifra por deslocamento conhecida foi a cifra de Cesar, que substitui cada letra do alfabeto deslocando-a 3 posições, equivalente a uma chave igual a 3.

Nessa parte foi implementada uma cifra de deslocamente com chave de tamanho 1, similar a cifra de Cesar, mas podendo escolher o deslocamento padrão para cada cifra. A seguir, serão apresentados os códigos da encriptação, decodificação e das tentativas de quebra por força bruta e por análise de frequência.

Oberservação: apesar de não ser uma boa prática de programação, foi utilizado using namespace std nos códigos para simplificar a escrita e deixar o código mais limpo.

1.1 Encriptação

Para a encriptação por deslocamento, como dito acima, foi feito um código simples, essa função recebe a chave, que pode ser qualquer número inteiro. O ideal é usar um número entre 1 e 25, já que o 0 apenas mantém o texto original. Mas, se o número digitado for maior, é feito o módulo desse valor por 26, então isso não interfere no funcionamento.

Além da chave, a função recebe o texto em claro, que deve estar sem pontuações, espaços, números ou caracteres especiais, e todo em letras minúsculas. Foi criado um código 14 separado, em Python, para fazer essa conversão do texto.

Para cada caractere, subtraímos o valor da letra 'a' (já que, em C++, os caracteres podem ser convertidos em inteiros). Esse valor é somado com a chave, depois tiramos o módulo 26 e, por fim, somamos novamente com 'a' para obter o novo caractere. Esse novo caractere é adicionado à string do ciphertext. Por fim, a função retorna a mensagem cifrada.

Abaixo, podemos conferir o código da função de encriptação por deslocamento:

```
1 string cript_desloc(string& plaintext, int k){
2     string ciphertext;
3     for(char c : plaintext){
4         ciphertext += (c-'a'+k) % 26 + 'a';
5     }
6     return ciphertext;
7 }
```

Listing 1: Função de encriptação por deslocamento

1.2 Decriptação

Para decriptografar a mensagem, basta receber a chave e a mensagem cifrada, e subtrair o valor da chave de cada caractere. Como esse valor subtraído pode ser negativo, depois subtrair o valor de 'a', assim como fizemos na encriptação, devemos somar 26 e então subtrair a chave. Dessa forma, ao tirar o módulo por 26, garantimos que o valor não fique negativo. Por fim, somamos novamente 'a' para obter o caractere original do texto em claro.

A seguir, é apresentado o código da função de decriptação por deslocamento:

```
string descript_desloc(string& ciphertext, int k){
    string plaintext;
    for(char c : ciphertext){
        plaintext += (c-'a'+26-k) % 26 + 'a';
    }
    return plaintext;
}
```

Listing 2: Função de decriptação por deslocamento

1.3 Quebra Por Ataque De Força Bruta

Como a cifra por deslocamento utilizada tem apenas um deslocamento padrão para todos os caracteres, o código de quebra por força bruta precisa apenas testar as 25 possíveis chaves restantes na função de descriptografia por deslocamento. A 26ª chave seria $k = 0$, ou seja, a mensagem resultaria no próprio *plaintext*, por isso ela não é testada. A complexidade dessa função é $O(25 \times n)$, onde n é o tamanho do texto cifrado.

Nesse caso de quebra, dependemos de um humano (ou de algum sistema automatizado treinado para reconhecer padrões linguísticos) para escolher a opção que faz sentido.

A seguir, podemos conferir o código da função de quebra da cifra por deslocamento, que utiliza a função de descriptografia apresentada anteriormente.

```
void quebraBruteForces_desloc(string& ciphertext){
    // brutando para descobrir o plaintext, com todos as 25
       chaves possiveis
    for(int i = 25; i > 0; i--){
        string plaintext = descript_desloc(ciphertext, i);
        cout << "Chave␣:␣" << 26 - i << "␣->␣" << plaintext << "\
           n";
    }
}
```

Listing 3: Função de quebra por força bruta da cifra por deslocamento

1.4 Quebra Por Análise da Distribuição de Frequência

Nesta última parte da cifra por deslocamento, o objetivo é a quebra da cifra usando uma tabela de distribuição de frequência dos caracteres em português. Neste trabalho, foi utilizada a tabela [1], sugerida na lista. Como dito anteriormente, as possibilidades da cifra escolhida são poucas, com baixo custo computacional e fácil de verificar para um humano. Por isso, nesse caso, esse tipo de quebra não apresenta uma performance significativamente melhor do que o ataque por força bruta, mas podemos observar que a mensagem correta costuma aparecer com mais frequência entre as primeiras opções testadas.

No código apresentado abaixo, foi armazenada uma string com os caracteres ordenados da maior para a menor frequência, conforme a tabela. Em seguida, é contabilizada a frequência de cada caractere na mensagem cifrada, e eles são ordenados em ordem decrescente. O caractere mais frequente da mensagem cifrada é então comparado com o mais frequente da tabela, obtendo-se uma chave, e a função de decriptação é utilizada para imprimir a resposta. Esse processo se repete, comparando, em ordem, o caractere mais frequente da mensagem cifrada com os outros da tabela.

```cpp
void quebraDistrFreq_desloc(string &ciphertext){
    // alfabeto ordenado por frequencia
    string alphFreq = "aeosirdntcmuplvgbfqhjzxkwy";
    vector<int> frequence(26, 0);
    for(auto c : ciphertext){
        frequence[c-'a']++;
    }

    // vetor para ordenar os caracteres do ciphertext por maior
       frequencia
    vector<pair<char, int>> ordChar;
    for(int i = 0; i < 26; i++)
        ordChar.push_back({'a' + i, frequence[i]});

    sort(ordChar.begin(), ordChar.end(),
        [](const pair<char, int>& a, const pair<char, int>& b) {
            return a.second > b.second;
        });

    // brutando para descobrir o plaintext, com todas as 26
       chaves possiveis,
    // mas seguindo a ordem da maior frequencia
    for(int i = 0; i < 26; i++){
        int k = (ordChar[0].first + 26 - alphFreq[i]) % 26;
        string plaintext = descript_desloc(ciphertext, k);
        cout << "Chave␣:␣" << k << "␣->␣" << plaintext << "\n";
    }
}
```

Listing 4: Função de descriptografia por análise de frequência

A complexidade, nesse caso, é equivalente à do brute force, $O(26 \cdot |ciphertext|)$.

1.5 Análise dos Resultados

Por causa da simplicidade da cifra por deslocamento implementada, os testes realizados mostraram que, mesmo com tentativas básicas de quebra, como a força bruta ou a análise de frequência, é possível recuperar o texto original com relativa facilidade. Em todos os testes realizados, a mensagem correta aparecia entre as opções sugeridas, e na análise de frequência, a opção correta era sempre uma das primeiras, demonstrando que esse tipo de implementação possui uma segurança muito fraca.

No livro texto utilizado [2], autor levanta o exemplo do *One-Time Pad*, que é considerada uma cifra perfeita. Nesse caso, uma chave realmente aleatória, do mesmo tamanho da mensagem e utilizada apenas uma vez, torna a quebra impossível, já que o *ciphertext* resultante é completamente aleatório e não revela nenhum padrão que possa ser analisado. No entanto, as dificuldades relacionadas ao armazenamento e à distribuição de uma chave tão grande tornam esse método impraticável em muitas situações reais.

Conclui-se que chaves maiores aumentam a segurança desse tipo de cifra, mas mesmo assim, isoladamente, elas não são suficientes para garantir proteção. Na prática, é necessário combinar diferentes métodos ou adotar sistemas criptográficos mais complexos para alcançar um nível de segurança confiável.

2 Quebrando Cifra Por Transposição

As cifras por transposição reordenam o texto sem alterar os caracteres, apenas mudando a posição deles. Neste trabalho, foi implementada a cifra de transposição com matriz. Para isso, primeiro foi escolhida uma chave, a qual o tamanho foi fixado como o a raiz quadrada do tamanho da mensagem, visto que foi autorizado fixar o tamanho da chave para facilitar a tentativa de quebra. E após escrever a mensagem na matriz, preenchendo linha por linha sequencialmente, as colunas são ordenadas com base na ordenação da chave em ordem alfabética. Por fim, a matriz é transposta, obtendo o ciphertext, o que pode ser feito lendo as colunas sequencialmente, ao invés das linhas. Para decifrar a mensagem, basta fazer o processo inverso.

Na implementação de criptografia e descriptografia feita, pode ocorrer repetição de letras do alfabeto na chave sem alterar o código, mas isso foi evitado ao máximo, sendo repetido apenas quando o tamanho da chave é maior do que o alfabeto.

2.1 Encriptação

Para a encriptação, foi criado inicialmente um vetor de strings, onde cada string representa uma coluna da matriz de transposição. Ou seja, esse vetor já corresponde à matriz transposta. Em seguida, para ordenar as colunas de acordo com a chave, foi criado um vetor auxiliar do tipo pair<char, int>, onde o primeiro elemento representa o caractere da chave na posição indicada pelo seu segundo elemento. Como essa ordenação utiliza o índice da posição do caractere, a presença de letras repetidas na chave não interfere no processo de encriptação ou decriptação. Após ordenar esse vetor auxiliar, obtém-se a nova ordem das colunas. Por fim, cada coluna é adicionada, seguindo essa nova ordem, à string final, que corresponde ao ciphertext, sendo esse o retorno da função.

```
string criptTransposicao(string k, string plaintext){
    //gerador de numero aleat rio
    mt19937 rng(chrono::steady_clock::now().time_since_epoch().
       count());

    // tamanho da chave
    int keySz = (int)k.size();

    // organizando o plaintext na matriz j  transposta
    // cada string do vetor representa uma coluna
    vector<string> matriz(keySz, "");
    for(int i=0; i < (int)plaintext.size(); i++){
        matriz[i%keySz] += plaintext[i];
    }

    // ordenando as colunas pela chave
    vector<pair<char, int>> ordKey;
    for(int i=0; i < keySz; i++) ordKey.emplace_back(k[i], i);
    sort(ordKey.begin(), ordKey.end());

    // criando o cyphertext
    string ciphertext = "";
    for(auto [c, pos] : ordKey){
```

```
23          ciphertext += matriz[pos];
24          // quando a palavra eh menor do que o padrao, completar
               com um caracter aleatorio
25          if((int)matriz[pos].size() < ((int)plaintext.size()+keySz
               -1)/keySz)
26              ciphertext += 'a' + (rng()%26);
27      }
28      return ciphertext;
29  }
```

Listing 5: Função de encriptação por transposição

2.2 Decriptação

Na decriptação, a função começa ordenando as colunas pela chave, da mesma forma que foi feito na encriptação. Em seguida, é criada uma *string* com o mesmo tamanho do *plaintext* original, preenchida inicialmente com o caractere 'a'. A substituição correta dos caracteres é feita por dois laços: o externo percorre os índices da chave, e o interno percorre os elementos de cada coluna.

Como a matriz utilizada na encriptação foi transposta, e o vetor com a ordem das colunas (`ordKey`) já está ordenado, foi possível reconstruir a mensagem original diretamente. A ideia é que a posição [i][j] da matriz original (antes da transposição) corresponde à posição [j][ordKey[i].second]. Como a matriz em si não foi reconstruída, a equivalência foi feita diretamente sobre a *string*: acessar [i][j] na matriz transposta equivale a [i ∗ tamanhoDaColuna + j], enquanto acessar [j][ordKey[i].second] na matriz original equivale a [ordKey[i].second + j ∗ tamanhoDaChave]. Isso permite substituir corretamente os caracteres do *ciphertext* para recuperar o *plaintext*.

```
1   string descriptTransposicao(string k, string ciphertext){
2       // tamanho da chave
3       int keySz = (int)k.size();
4
5       // ordenando as colunas pela chave
6       vector<pair<char, int>> ordKey;
7       for(int i=0; i < keySz; i++) ordKey.emplace_back(k[i], i);
8       sort(ordKey.begin(), ordKey.end());
9
10      //decifrando o ciphertext
11      int textsz = (int)ciphertext.size()/keySz;
12      string plaintext((int)ciphertext.size(), 'a');
13      for(int i=0; i < keySz; i++){
14          for(int j=0; j < textsz; j++){
15              plaintext[ordKey[i].second+keySz*j] = ciphertext[i*
                   textsz + j];
16          }
17      }
18
19      return plaintext;
20  }
```

Listing 6: Função de decriptação por transposição

2.3 Quebra Por Ataque De Força Bruta

A cifra por transposição utilizada, num ataque ingênuo de força bruta sabendo o tamanho da chave, possui complexidade de $|chave|!$, pois é necessário gerar todas as permutações possíveis da chave. Para construir cada mensagem a partir de uma dessas chaves, o custo computacional da função utilizada é aproximadamente $|mensagem| + |chave| \cdot \log |chave|$, sendo esse segundo termo normalmente desprezível.

Sabendo disso, se o tamanho da chave não fosse fixo, teríamos que testar esse código para todos os tamanhos possíveis de chaves, resultando na seguinte complexidade total:

$$\left(\sum_{i=1}^{|chave|} i! \right) \cdot |mensagem|$$

Percebe-se que esse valor cresce muito rapidamente, tornando o cálculo inviável na prática. Por isso, o código de quebra da cifra de transposição por força bruta foi implementado, mas ele foi testado apenas com chaves pequenas, para evitar sobrecarga no computador.

A quebra da cifra foi dividida em duas funções:

- A função `ordem`, que recebe o tamanho da chave e o `ciphertext`, gera todas as permutações possíveis da chave representadas por um vetor de inteiros de `0` a `keySz - 1`, utilizando a função `next_permutation`. Com a complexidade de $|chave|!$.
- A função `quebra`, que recebe uma dessas permutações e reconstrói o `plaintext` conforme a lógica da descriptografia, aplicando o vetor `sequencia` como chave. Com a complexidade de $|ciphertext|$.

Estes códigos podem ser conferido abaixo:

```
void ordem(int keySz, string& ciphertext){
    vector<int> sequencia;
    for(int i = 0; i < keySz; i++) sequencia.push_back(i);
    do{
        quebra(sequencia, ciphertext, keySz);
    } while(next_permutation(sequencia.begin(), sequencia.end()))
       ;
}
```

Listing 7: Função que gera todas as possíveis sequências

```
void quebra(vector<int>& sequencia, string& ciphertext, int keySz
   ){
    // ordenando as colunas pela chave
    vector<pair<int, int>> ordKey;
    for(int i = 0; i < keySz; i++) ordKey.emplace_back(sequencia[
       i], i);
    sort(ordKey.begin(), ordKey.end());

    // quebrando o ciphertext
    int textsz = (int)ciphertext.size() / keySz;
    vector<char> plaintext((int)ciphertext.size());
    for(int i = 0; i < keySz; i++){
```

```
11          for(int j = 0; j < textsz; j++){
12              plaintext[ordKey[i].second + keySz * j] = ciphertext[
                    i * textsz + j];
13          }
14      }
15
16      for(char c : plaintext) cout << c;
17      cout << endl;
18  }
```

Listing 8: Função de quebra da cifra com uma sequência

2.4 Quebra Por Análise da Distribuição de Frequência

Para a quebra por análise de distribuição de freqüencia, o código foi divido em várias partes, que serão explicadas individualmente. A ideia principal do código foi precalcular a probabilidade de cada dupla e trio de colunas, atráves das tabelas de distribuição de frequência dos dígrafos e trígrafos, e formar um arranjo entre essas colunas, atráves da combinação mais provável entre elas.

Inicialmente, foi recebido o ciphertext e então montada a matriz já transposta dele, mas com cada caracter valendo de 0 a 26, para facilitar os calculos depois. Como o tamanho da chave foi fixado como $\sqrt{|ciphertext|}$, montar essa matriz é simples e o código pode ser conferido a seguir.

```
1  // monta a matriz de transposi  o do ciphertext
2  vector<vector<int>> matriz_transposicao(int keySz, string
       ciphertext){
3      int textSz = (int)ciphertext.size()/keySz;
4      vector<vector<int>> ciphermatriz(textSz, vector<int>(keySz));
5      //complexidade O(cipherText.size())
6      for(int i=0; i < keySz; i++){
7          for(int j=0; j < textSz; j++){
8              ciphermatriz[j][i] = ciphertext[i*textSz+j]-'a';
9          }
10     }
11     return ciphermatriz;
12 }
```

Listing 9: Função que gera a matriz de transposição do ciphertext

Depois, para precalcular a probabilidade de cada dupla e trio de colunas, foram construídas duas matrizes, com os dados obtidos atráves da tabela [1]. A primeira matriz tem duas dimensões e armazena a frequência média de cada um dos dígrados possíveis, tendo tamanho 26*26. Ela foi armazenada num arquivo chamado tabela_digrafos.hpp, que foi importado no início do arquivo do código da quebra. E a segunda tem três dimensões e armazena a freqüencia média dos trigrafos, mas como no arquivo só contia o valor dos 40 mais frequentes, para o resto foi atribuido o valor 0. Essa matriz tem tamanho 26*26*26 e foi armazenada num arquivo chamado tabela_trigrafos.hpp. Ela também foi importada no início do arquivo do código da quebra e, no início da main(), tem que chamar a função inicializar_trigrafos() para preencher na matriz os valores dos 40 trigrafos mais frequentes.

Com essas matrizes, foram contruídas duas funções para precalcular as probabilidades. A primeira, precalcula a probabilidade de cada dupla de colunas. Apesar de estar sendo chamado de probabilidade, esse valor corresponde apenas a soma da frequencia de cada digrafo que aparece na dupla de colunas. Como parametros, a função recebe o tamanho da chave, das colunas e a matriz transposta, e retorna uma matriz quadrada, sendo n = $|chave|$. A complexidade dessa função é de $O(|chave| \cdot |ciphertext|)$.

```
// precalcula a probabilidade de cada dupla de colunas
vector<vector<double>> precalc_digrafos_prob(int keySz, int
   columnSz, vector<vector<int>>& ciphermatriz){
    vector<vector<double>> dupla_probabilidade(keySz, vector<
       double>(keySz, 0));
    //complexidade O(keySz^2*columnSz) == O(|cipherText|*keySz)
    for(int i=0; i < keySz; i++){
        for(int j=0; j < keySz; j++){
            if(i == j) continue;
            for(int k=0; k < columnSz; k++){
                dupla_probabilidade[i][j] += digrafos[
                   ciphermatriz[k][i]][ciphermatriz[k][j]];
            }
        }
    }
    return dupla_probabilidade;
}
```

Listing 10: Função que gera a matriz com a probabilidade entre cada dupla de colunas

A segunda função precalcula a probabilidade de cada trio de colunas, similar a das duplas, mas agora com três colunas, então a matriz é cúbica, e a complexidade é de $O(|chave|^2 \cdot |ciphertext|)$, gastando mais tempo e memória. O seu código é apresentado abaixo.

```
// precalcula a probabilidade de cada trio de colunas
vector<vector<vector<double>>> precalc_trigrafos_prob(int keySz,
   int columnSz, vector<vector<int>>& ciphermatriz){
    vector<vector<vector<double>>> trio_probabilidade(keySz,
       vector<vector<double>>(keySz, vector<double>(keySz, 0)));
    // complexidade O(keySz^3*columnSz) == O(|cipherText|*keySz
       ^2)
    for(int i=0; i < keySz; i++){
        for(int j=0; j < keySz; j++){
            if(i == j) continue;
            for(int k=0; k < keySz; k++){
                if(k == i || k == j) continue;
                for(int z=0; z < columnSz; z++){
                    trio_probabilidade[i][j][k] += trigrafos[
                       ciphermatriz[z][i]][ciphermatriz[z][j]][
                       ciphermatriz[z][k]];
                }
            }
        }
    }
    return trio_probabilidade;
```

```
}
```

Listing 11: Função que gera a matriz com a probabilidade entre cada trio de colunas

A parte final do código foi dividida em três funções, sendo que a terceira já foi apresentada anteriormente na quebra por força bruta (Listing 8). Essa primeira função tem como responsabilidade chamar a função que retorna as 10 combinações mais prováveis da chave (Listing 13), com base nas tabelas de frequências, e em seguida, para cada uma dessas combinações, aplicar a função de quebra para cada um dos $|chave|$ deslocamentos cíclicos possíveis desse arranjo.

A complexidade dessa função é dada pela soma do custo da geração dos arranjos e do custo das chamadas da função de quebra, resultando em:

$$10 \cdot |chave|^2 + 10 \cdot |chave| \cdot |ciphertext|$$

Como o segundo termo domina a expressão, a complexidade da função pode ser aproximada por $O(10 \cdot |chave| \cdot |ciphertext|)$.

O código da primeira função é apresentado a seguir, com a utilização de um `typedef` para simplificar a chamada da função.

```
typedef tuple<double, vector<int>, vector<int>> combo;

// imprimi todos os cyclic shifts possiveis das 10 combinacoes mais provaveis
// complexidade O(10*keySz*|ciphertext|)
void quebraFreq(int keySz, string& ciphertext, vector<vector<double>>& dupla_prob, vector<vector<vector<double>>> trio_prob){
    // probabilidade do arranjo, vetor das colunas na ordem colocada, vetor das colunas em ordem crescente
    vector<combo> listaArranjos = {{0, {0}, {0}}};
    listaArranjos = novaLista(keySz, listaArranjos, dupla_prob, trio_prob);
    // percorrendo os 10 melhores arranjos
    for(auto [prob, ordem, visited] : listaArranjos){
        // chamando a funcao quebra para cada posicao do vetor ordem sendo a inicial
        for(int i=0; i < keySz; i++){
            quebra(ordem, ciphertext, keySz);
            //cyclic shifted do vetor ordem
            rotate(ordem.begin(), ordem.begin() + 1, ordem.end());
        }
    }
}
```

Listing 12: Função que gera a matriz com a probabilidade entre cada trio de colunas

Como o código de força bruta fica muito grande, para encontrar os arranjos mais prováveis da chave de uma cifra por transposição, foi implementada uma função recursiva que constrói sequências de colunas com base na tabela de frequências de dígrafos e trígrafos. A entrada da função é uma lista de tuplas, onde cada tupla contém:

- uma probabilidade acumulada (valor `double`),

- uma sequência parcial da chave (`vector<int>`),
- e um vetor com os índices já utilizados (`vector<int>`), ordenado.

A função verifica, inicialmente, se a sequência atual tem o mesmo tamanho da chave. Se tiver, retornamos essa lista como resultado final. Caso contrário, criamos uma nova lista para armazenar extensões possíveis dessas sequências.

Para cada tupla da lista atual, percorremos todos os possíveis índices de coluna ainda não utilizados. Se um índice ainda não estiver na sequência, criamos uma nova sequência com esse valor adicionado. A nova probabilidade é calculada somando-se:

- a probabilidade anterior;
- a probabilidade da transição entre a última coluna e a nova (com base na tabela de dígrafos);
- e, se já houver pelo menos duas colunas na sequência, também a probabilidade da transição envolvendo as duas últimas e a nova (com base na tabela de trígrafos).

Essas novas tuplas são armazenadas em uma lista auxiliar. Após gerar todas as possibilidades, ordenamos essa lista pela probabilidade (em ordem decrescente) e selecionamos apenas os 10 arranjos mais promissores. A função é então chamada recursivamente com essa nova lista, continuando o processo de construção das sequências até que todas tenham o tamanho da chave.

Esse método permite reduzir significativamente o número de permutações testadas, mantendo apenas as mais prováveis. Abaixo é apresentado o código dessa função.

```
1  typedef tuple<double, vector<int>, vector<int>> combo;
2
3  // cria os 10 chaves mais provaveis
4  vector<combo> novaLista(int keySz, vector<combo>& lastList,
       vector<vector<double>>& dupla_prob, vector<vector<vector<
       double>>>& trio_prob){
5      // verifica se o tamanho da possivel chave eh igual ao
           tamanho da chave
6      if((int)(get<1>(lastList[0])).size() == keySz) return
           lastList;
7      vector<tuple<int, vector<int>, vector<int>>> newList;
8      for(auto [probabilidade, ordem, visited] : lastList){
9          for(int i=0; i < keySz; i++){
10             // verifica se o elemento nao esta presente na lista
11             if(find(visited.begin(), visited.end(), i) == visited
                   .end()){
12                 int szOrdem = (int)ordem.size();
13                 // soma a probabilidade da nova coluna com a
                       ultima coluna
14                 double new_prob = probabilidade + dupla_prob[
                       ordem[szOrdem-1]][i];
15                 if(szOrdem > 1) new_prob += trio_prob[ordem[
                       szOrdem-2]][ordem[szOrdem-1]][i];
16                 vector<int> new_ordem = ordem;
17                 new_ordem.push_back(i);
```

```
                    vector<int> new_visited = visited;
                    new_visited.push_back(i);
                    sort(new_visited.begin(), new_visited.end());
                    newList.emplace_back(new_prob, new_ordem,
                        new_visited);
                }
            }
        }
        sort(newList.begin(), newList.end(), [](const auto& a, const
            auto& b) {
            return get<0>(a) > get<0>(b);  // ordem descrescente
        });
        //mantem os 10 primeiros arranjos com maior probabilidade de
            estarem certos
        lastList = vector<combo>(newList.begin(), newList.begin()+min
            (10, (int)newList.size()));
        return novaLista(keySz, lastList, dupla_prob, trio_prob);
}
```

Listing 13: Função que gera a matriz com a probabilidade entre cada trio de colunas

2.5 Testes e Análises de Resultados

Para verificar a eficácia do código de quebra da cifra por transposição, foram testadas quatro mensagens diferentes. O código desenvolvido recebe como entrada o *plaintext* original e todas as respostas geradas pela função, e calcula o *matching* (Listing 16) (proporção de caracteres corretamente posicionados) para cada uma delas , selecionando o maior *matching*.

Nos dois primeiros exemplos, o código foi capaz de recuperar exatamente a mensagem original, atingindo 100% de *matching*. Os exemplos foram:

- **Chave:** `zuosvxi`
- **Mensagem original:**
  olamundoeumprazerveloaquivamoscomecaryogfj
- **Resultado:**
  olamundoeumprazerveloaquivamoscomecaryogfj
- **Chave:** `kqolmpnecr`
- **Mensagem original:**
  desenvolversolucoescriativaseeficientespararesolverproblemascomplexosesempreum desafiossvyh
- **Resultado:**
  desenvolversolucoescriativaseeficientespararesolverproblemascomplexosesempreum desafiossvyh

No terceiro exemplo, no entanto, o código atingiu um *matching* de apenas 51,47%, e não conseguiu recuperar a mensagem completamente. Ainda assim, a estrutura da mensagem permite identificar o início correto da frase já no primeiro bloco, o que demonstra que a maior parte da reorganização foi bem conduzida. A mensagem original era:

- **Chave:** `qdoziuajgzrhn`
- **Mensagem original (início):**
  aoenfrentardesafiosinesperadosacriatividadesetornaumaferramentaessencialparaencontrarsolucoesinovadoraseaomesmotempomanterofoconosobjetivoseresultadosdesejadosozbyotsjwq
- **Resultado obtido:**
  aoenfredentarsafiosreinespadosacivriatidadesemutornaaferrasementasencianelparacontraocrsoluesinovesadoraaomesmmootempanterosofoconobjetisevoserultadoajsdesedosozbqwyotsj

Observa-se que, ao reorganizar manualmente os blocos iniciais da mensagem — por exemplo, movendo as colunas da posição 7 e 8 para o final — é possível recuperar facilmente a estrutura correta da frase. Isso reforça a ideia de que uma abordagem híbrida, combinando análise automática com intervenção humana, pode melhorar significativamente os resultados em casos mais difíceis.

Por último, foi realizado mais um teste com uma mensagem maior:

- **Chave:** `zdjxqluvgfkenbmry`
- **Mensagem original (início):**
  acapacidadederesolverproblemascomplexosestadiretamenterelacionadaahabilidadedeanalisarsituacoessobdiferentesperspectivasemcontextosondeacriatividadesealiaaoraciociniologicosurgemsolucoesinovadorascapazesdetransformardesafiosemoportunidadesdecrescimentoeaprendizadocontinuo
- **Resultado obtido:**
  cidadedeseaorcaparoblemasmolpcverpestadiremaletexoslacionadhanaatereadedeanasibalilidacoessobfiredsitusperspecviratententextosoedsanemcoividadeslacieriatciociniogoailaoragemsolucsecioosurorascapasendzovadsformardasefetranoportunidaiedosemescimentaespodecrzadocontunroiendi

Neste caso, o valor de *matching* foi de apenas 33,33%, indicando que o algoritmo não conseguiu reconstruir corretamente a maior parte da mensagem. Ainda assim, é possível identificar blocos de palavras reconhecíveis, e com a chave correta ou ajustes manuais nos blocos transpostos, a recuperação completa da mensagem seria viável.

2.5.1 Conclusão

De forma geral, o comportamento do código na etapa de transposição correspondeu às expectativas. A parte final da quebra por análise de frequência foi estruturada de maneira heurística, baseada em tabelas de trígrafos e dígrafos da língua portuguesa, o que permitiu reduzir bastante o número de permutações testadas. No entanto, a complexidade do código acabou sendo dominada pelo pré-cálculo da matriz de probabilidades entre tríos de colunas. Como essa matriz possui dimensão cúbica ($26 \times 26 \times 26$), e apenas os 40 trígrafos mais frequentes foram utilizados, essa etapa pode ser repensada em contextos com chaves maiores, seja retirando a análise por trígrafos ou otimizando sua estrutura. Ainda assim, neste trabalho, essa limitação foi ignorada.

O desempenho do código não foi perfeito, mas foi melhor do que o esperado: em alguns testes, a função foi capaz de recuperar 100% da mensagem original. Apesar de serem

consideradas apenas 10 possíveis chaves mais prováveis, a função aplica deslocamentos cíclicos a cada uma delas, o que aumenta significativamente a cobertura do espaço de busca. Isso garante que o código, mesmo com uma abordagem restrita, consiga encontrar a chave correta com frequência.

Uma possível melhoria para essa abordagem seria analisar, inicialmente, apenas o primeiro bloco da mensagem, de tamanho igual à chave. Isso permitiria detectar trocas de colunas dentro de cada bloco, e, com auxílio humano, ajustar manualmente os blocos transpostos, o que se mostrou uma tarefa simples em muitos exemplos testados.

Esses resultados evidenciam as limitações impostas pela simplicidade da heurística utilizada, mas também mostram a fragilidade da cifra por transposição implementada. Foi possível perceber que a segurança dessa cifra, da forma como foi aplicada neste trabalho, é bastante limitada. Mesmo sem experiência prévia em criptoanálise, em muitos casos a mensagem foi parcialmente ou totalmente recuperada, especialmente quando a chave era pequena ou a estrutura da mensagem facilitava a identificação de padrões. Isso demonstra que, isoladamente, essa cifra não é suficiente para garantir a segurança de uma mensagem.

Uma possível melhoria seria adotar uma abordagem híbrida, combinando a transposição com uma cifra por deslocamento. Além disso, a aplicação repetida da transposição (em duas ou mais camadas) também aumentaria consideravelmente a complexidade da quebra, tornando o processo mais resistente a ataques por força bruta ou heurísticas de frequência.

3 Códigos Extras

Nesta seção, são apresentados os códigos extras utilizados para testes, mas que não fazem parte da lista.

3.1 Formatação Inicial da Mensagem

Para fazer a formatação de uma mensagem, foi criado um código em `Python` que normaliza o texto, removendo espaços, acentos, números, pontuações e caracteres especiais, além de transformar todos os caracteres em letras minúsculas.

```python
import unicodedata
import re

def limpar_texto(texto):
    # normaliza e remove acentos
    texto = unicodedata.normalize('NFD', texto)
    texto = texto.encode('ascii', 'ignore').decode('utf-8')
    # converte para minusculo, remove espacos, pontuacoes e
       numeros
    texto = re.sub(r'[^a-zA-Z]', '', texto.lower())
    return texto
```

Listing 14: Função para limpeza de texto

3.2 Gerador de Chave Aleatório

Foi desenvolvido um gerador de chaves aleatórias em `C++`, onde a chave tem tamanho definido como o teto da raiz quadrada do tamanho do texto. Esse valor foi fixado para a quebra da cifra por transposição, mas pode ser facilmente ajustado dentro da função. A função recebe o tamanho do texto e calcula o comprimento da chave com base nesse valor.

Para garantir diferentes saídas a cada execução, a semente do gerador é baseada no tempo atual do sistema. Isso é feito por meio da função chrono::steady_clock :: now().time_since_epoch().count(), que retorna um valor numérico correspondente ao tempo decorrido desde a "época" do relógio. Esse valor é então utilizado como semente para o gerador Mersenne Twister mt19937, garantindo aleatoriedade nas saídas a cada execução.

O gerador rng é utilizado para embaralhar a string do alfabeto, que é repetida aleatoriamente até atingir o tamanho desejado da chave. Assim, a chave gerada possui caracteres aleatórios e não se repete em execuções consecutivas.

```cpp
string geradorChaveRandom(int textsz){
    // gerador de numero aleatorio
    mt19937 rng(chrono::steady_clock::now().time_since_epoch().
       count());

    // definindo tamanho da chave como o teto da raiz do tamanho
       do texto
    int keysize=1;
    for(int i=1; i*i < textsz; i++) keysize=i;
    keysize++;
```

```
9
10      string key="", alph = "abcdefghijklmnopqrstuvwxyz", newalph;
11
12      // criando a chave ordenando aleatoriamente o alfabeto
13      while((int)key.size() < keysize){
14          newalph = alph;
15          shuffle(newalph.begin(), newalph.end(), rng);
16          for(int i=0; i < min(26, keysize-(int)key.size()); i++){
17              key += newalph[i];
18          }
19      }
20      return key;
21  }
```

Listing 15: Função para gerar uma chave aleatoria

3.3 Cálculo de Matching entre Mensagens

Para comparar se as mensagens obtidas por meio da quebra por transposição eram semelhantes às originais, foi implementada uma função em C++ para calcular a similaridade (matching) entre duas strings. Trata-se de uma função simples, que apenas conta as posições nas quais os caracteres coincidem e divide esse valor pelo tamanho total da string, retornando um valor real entre 0 e 1. Quanto mais próximo de 1, mais precisa foi a quebra.

Essa função foi utilizada na análise dos resultados. No entanto, observa-se que, em alguns casos, mesmo com um valor de matching relativamente baixo, a mensagem ainda pôde ser compreendida corretamente.

```
1   double calcularMatching(const string& str1, const string& str2) {
2       int matches = 0;
3
4       for (size_t i = 0; i < str1.length(); i++) {
5           if (str1[i] == str2[i]) matches++;
6       }
7
8       // retorna o valor do matching (posicoes iguais / tamanho da
            string)
9       return (matches*1.0) / (double)str1.length();
10  }
```

Listing 16: Função para calcular o matching entre duas strings

References

[1] Rogério Reis. *Tabelas de Frequências na Língua Portuguesa*. Disponível em: https://www.dcc.fc.up.pt/~rvr/naulas/tabelasPT/. Acesso em: abril de 2025.

[2] William Stallings. *Cryptography and Network Security: Principles and Practice, Global Edition*. Pearson Education, Germany, 2022.