Algoritmos Elementares

Nesta seção são ilustrados algoritmos elementares sobre strings, com o intuito de ilustrar a manipulação, identificação e extração de informações de strings. As técnicas apresentadas servirão tanto para solidificar os conceitos fundamentais quanto para preprar o leitor para os algoritmos mais elaborados das seções subsequentes.

Palíndromos

Palíndromos são strings que são idênticas quando lidas tanto do início para o fim quanto do fim para o início (por exemplo, MUSSUM, SAIAS e HANNAH são palíndromos). Mais formalmente, um palíndromo `P` pode ser definido como

```
P[1..N] = "" | P[1..1] | P[2..N-1] and P[1] == P[N],
```

ou seja, strings vazias, strings com um único caractere ou strings resultantes da concatenação de um mesmo caractere no ínicio e no fim de um palíndromo resulta em palíndromos.

Uma maneira de se verificar se uma string `s` é ou não palíndromo é checar se o primeiro caractere coincide com o último, o segundo com o penúltimo, e assim por diante.

```
bool is_palindrome(const string& s)
{
    size_t N = s.size();

    for (size_t i = 0; i < N; ++i)
        if (s[i] != s[N - 1 - i])
            return false;

    return true;
}
```

Este algoritmo tem complexidade `O(N)`. Embora ele identifique corretamente se `s` é ou não um palíndromo, é possível torná-lo mais eficiente ao observar que só é necessário fazer tal verificação até a metade de `s` (pois, se `i >= N/2`, temos `i = N - 1 - j, j < N/2` e a comparação `s[i] == s[N - 1 - i]` equivale a comparação `s[N - 1 - j] == s[N - 1 - (N - 1 - j)]`, isto é, `s[N - 1 - j] == s[j], j < N/2`. Mesmo que a complexidade permaneça `O(N)`, a versão abaixo executa duas vezes mais rápido que a anterior.

```
bool is_palindrome2(const string& s)
{
    size_t N = s.size();

    for (size_t i = 0; i < N/2; ++i)
        if (s[i] != s[N - 1 - i])
            return false;

    return true;
}
```

Observe que `is_palindrome2()` identifica, corretamente, strings `s` de tamanho par e ímpar (verifique este fato observando como o algoritmo lida com os caracteres centrais da strings em ambos casos).

Histograma

Uma técnica, oriunda da estatística, que permite identificar características importantes de uma strings é o **histograma**, que consiste em uma mapeamento entre os caracteres do alfabeto e o número de ocorrências dos mesmos em uma string `s` dada. Por exemplo, para a string `"abacaxi"`, teríamos um histograma `h` com

```
h['a'] = 4
h['b'] = h['c'] = h['x'] = h['i'] = 1
h[c] = 0,   c not in "abcxi"
```

Há 3 técnicas para a construção de histogramas. A primeira delas é utilizar a classe `map` do C++, que permite uma construção bastante intuitiva e fácil de histogramas, tendo como revezes a quantidade de memória necessária (o que, em geral, não chega a ser um problema) e a complexidade dos acessos (`O(log N)` para leitura e escrita).

Abaixo segue uma implementação bastante sintética em C++11, com uso de `auto` e *range for*.

```
#include <map>

std::map<char, int> histogram(const string& s)
{
    std::map<char, int> h;

    for (auto c : s)
        ++h[c];

    return h;
}
```

Uma segunda forma de gerar o histograma é utilizar um *array* estático com 256 posições (que cobre todos os possíveis valores de um `char` em C/C++). Esta estratégia permite atualizar/consultar os valores em `O(1)`, mas a identificação dos caracteres com valores diferentes de zero é linear neste *array* (o que, na prática, geralmente não constitui problema).

```
#include <cstring>

void histogram(int h[256], const string& s)
{
    memset(h, 0, sizeof h);

    for (auto c : s)
        ++h[c];
}
```

Na implementação acima, `h` é um parâmetro de saída, que deve ser inicializado com zeros no início da rotina.

A terceira e última maneira é uma otimização, em espaço, da segunda. Aqui o vetor `h` deve ter o tamanho `M` do alfabeto, e os caracteres do alfabeto devem ser indexados de 0 a `M - 1`. Se o alfabeto for constituído das letras maiúsculas e minúsculas, estas indexação é feita de forma direta, em `O(1)`. Caso contrário,é preciso buscar o caractere no alfabeto em `O(N)` (ou `O(log N)`, se o alfabeto estiver ordenado). Neste cenário a perda de performance é compensada pela redução da memória necessária (esta é a abordagem mais econômica em termos de memória).

Abaixo um exemplo onde o alfabeto é composto pelas 26 letras maiúsculas.

```
void histogram(int h[26], const string& s)
{
    memset(h, 0, sizeof h);

    for (auto c : s)
        ++h[c - 'A'];
}
```

Mapas, Filtros e Reduções

Mapas, filtros e reduções são técnicas de programação funcional que permitem alterar os elementos de um vetor, gerar um novo vetor a partir da seleção de elementos específicos de um vetor dado ou gerar um único objeto ou elemento a partir dos elementos de um vetor. Sendo uma string um vetor de caracteres, estas técnicas podem ser adaptadas para o contexto da manipulação de textos e letras.

Mapas

Um **mapa** (ou mapeamento) consiste em uma função `m_f: S_N -> S_N`, onde `S_N` é o conjunto de todas as strings de tamanho `N`. e `f: A -> A` é uma função cujo domínio é o alfabeto `A`, tal que se `y = m_f(s)`, então `y[i] = f(s[i])`. Em termos mais simples, `m` mapeia cada caractere de `s` de acordo com a função `f`.

Por exemplo, se `A` é formado pelas letras alfabéticas maiúsculas e minúsculas e `f` é a função `toupper()`, o mapeamento `m_f` tornaria maiúsculas todas as letras de uma string `s` dada. Uma implementação possível deste mapeamento é dada abaixo: a função `uppercase()` é utilizada apenas para corrigir o retorno da função `toupper()` e tornar o mapeamento reutilizável. O nome `smap` foi utilizado apenas para não conflitar com a classe `map` da biblioteca padrão do C++.

```
#include <cctype>

char uppercase(char c)
{
    return (char) toupper(c);
}

void smap(string& s, char (*f)(char))
{
    for (size_t i = 0; i < s.size(); ++i)
        s[i] = f(s[i]);
}
```

Outra forma de implementar um mapeamento em C++ é através da função `transform()`, que está declarado na biblioteca `algorithm`. Deve-se atentar, porém, que ela não suporta *arrays* primitivos, apenas containers. Os dois primeiros parâmetros indicam o intervalo a ser considerado, o terceiro é o método de inserção no container que conterá o resultado, e o último é um operador unário que recebe um elemento do container e retorna outro de mesmo tipo.

Abaixo uma implementação da Cifra de César usando a função `transform()`. Observe o uso de lambda no quarto parâmetro.

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
    string text { "cesar cipher xyz" };
    string res;

    transform(text.begin(), text.end(), back_inserter(res), [](char c)
        {
            return c == ' ' ? c : (((c - 'a') + 3) % 26) + 'a';
        }
    );

    cout << res << endl;

    return 0;
}
```

Filtros

Um **filtro** `f_p : S_N -> S_M` consiste em uma função que gera uma string de tamanho `M <= N`, a partir de uma string de tamanho `N`, através da seleção dos `M` caracteres que cujo predicado `p: A -> Bool` retorna verdadeiro.

Abaixo segue uma implementação simples que seleciona as vogais de uma string dada.

```
#include <cctype>

bool is_vowel(char c)
```

```
{
    const string vowels { "aeiou" };

    return vowels.find(tolower(c)) != string::npos;
}

string filter_vowels(const string& s)
{
    string v = "";

    for (auto c : s)
        if (is_vowel(c))
            v += c;

    return v;
}
```

Outra maneira de implementar o mesmo código é utilizar a função `copy_if()`, que tem sintaxe semelhante ao da função `transform()`. A função `remove_copy_if()` tem comportamento análogo, mas copia os caracteres que negarem o predicado.

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
    string text { "cesar cipher xyz" };
    string res;

    copy_if(text.begin(), text.end(), back_inserter(res), [](char c)
        {
            const string vowels { "aeiou" };

            return vowels.find(c) != string::npos;
        }
    );

    cout << res << endl;

    return 0;
}
```

Reduções

Uma **redução** `r_b : S_N -> T` gera um elemento do tipo `T` através da aplicação sucessiva, do operador binário `b`, da esquerda para a direita, em cada elemento de `s`, tendo como operador esquerdo inicial o valor passado como parâmetro.

Uma implementação direta da redução que soma os dígitos contidos na string `s` é dada abaixo.

```
int sum(const string& s)
{
    int s = 0;

    for (auto c : s)
        s += (c - '0');

    return s;
}
```

Em C++, é possível utilizar a função `accumulate()` para reduções. O mesmo código acima pode ser reimplementado da seguinte maneira.

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
    string text { "12345" };
```

```
    int res = accumulate(text.begin(), text.end(), 0, [](int L, int R)
        {
            return L + (R - '0');
        }
    );

    cout << res << endl;

    return 0;
}
```

Observe que o uso das funções `transform(), copy_if()` e `accumulate()` evitam o uso de laços de repetição.

Tabelas de substituição

Uma **tabela de substituição** é uma aplicação prática de um mapeamento. Ela é definida por uma função `f: A -> A` que mapeia cada caractere do alfabeto `A` em outro caractere do mesmo alfabeto. Sistemas criptográficos mais simples utilizam tabelas de substituição, como a Cifra de César, já citada, e a criptografia baseada em `xor`.

Abaixo segue um exemplo de criptografia `xor`. A tabela de substituição `table` é precomputada e é utilizado um mapeamento para cifrar e decifrar as mensagens.

```
#include <iostream>
#include <algorithm>

using namespace std;

#define MAX 256

char table[MAX];

void fill_table(char key)
{
    for (int i = 0; i < MAX; ++i)
        table[i] = i ^ key;
}

string cipher(const string& text)
{
    string res;

    transform(text.begin(), text.end(), back_inserter(res), [](char c)
        {
            return table[(int) c];
        }
    );

    return res;
}

string decipher(const string& c)
{
    return cipher(c);
}

int main()
{
    fill_table(0x3B);

    string message {  "Xor cipher example" };
    string c =  cipher(message);

    printf("c =");
    for (auto t : c)
        printf(" %02x", t);
    printf("\n");

    string d =  decipher(c);

    printf("d = [%s]\n", d.c_str());

    return 0;
}
```

Tokenização

Tokenização é o processo de quebra de um texto em palavras, frases, símbolos, etc, denominados **tokens**. Para realizar a tokenização, é necessário primeiramente definir precisamente um token, e esta definição depende do contexto onde os tokens serão utilizados.

Uma abordagem simples é definir uma lista de caracteres delimitadores (como espaços em branco, pontuações, etc) e dividir a string a cada ocorrência destes. Por exemplo, a frase "O rato roeu a roupa do rei de Roma." seria dividia em 9 tokens: "O", "rato", "roeu", "a", "roupa", "do", "rei", "de", "Roma".

A linguagem C oferece uma função para tokenização na biblioteca `string.h`, denominada `strtok()`. Ela recebe como primeiro parâmetro um ponteiro para a string a ser tokenizada, e como segundo parâmetro uma lista de delimitadores. Ela retorna um ponteiro para o início do próximo token, ou `NULL`, caso não existam mais tokens.

Esta função tem um comportamento incomum, devido três aspectos importantes:

1. ela mantém, internamente, um ponteiro para o início do próximo token. Desta forma, para obter vários tokens de uma mesma string, as chamadas subsequentes à primeira devem receber `NULL` como primeiro parâmetro (este comportamento faz com que esta função não seja segura num contexto *multithread*);
2. esta função altera o parâmetro de entrada, escrevendo o caractere `\0` nas posições onde são encontrados delimitadores. Assim, se for necessário preservar o conteúdo original da string, deve ser passada uma cópia da mesma como primeiro parâmetro;
3. a função `strtok()` não retorna tokens vazios (isto é, de tamanho zero).

O código abaixo ilustra o uso desta função para extrair os tokens de um CPF.

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

int main()
{
    char cpf[] = "123.456.789-10";
    char *token = NULL;

    token = strtok(cpf, ".");
    printf("token = [%s]\n", token);    /* token = "123" */

    token = strtok(NULL, ".");
    printf("token = [%s]\n", token);    /* token = "456" */

    token = strtok(NULL, "-");
    printf("token = [%s]\n", token);    /* token = "798" */

    token = strtok(NULL, "\0");
    printf("token = [%s]\n", token);    /* token = "10" */

    printf("cpf = %s\n", cpf);          /* cpf = "123" */

    return 0;
}
```

A mesma biblioteca oferece uma versão *thread safe* de `strtok()`, denominada `strtok_r()`. Nesta versão, há um terceiro parâmetro, que é utilizado para memorizar a posição do próximo token.

Em C++, a função `getline()` da biblioteca `string` pode ser utilizada para tokenização. Ela recebe como primeiro parâmetro um fluxo de entrada (`cin`, um arquivo, um fluxo de caracteres, etc), como segundo parâmetro a string que conterá o token identificado e, como terceiro parâmetro, o caractere delimitador (se omitido, o caractere `\0` será considerado o delimitador). Esta função é útil quando há apenas um delimitador e os tokens são lidos diretamente da entrada.

O código de extração de tokens do CPF por ser reescrito em C++ da seguinte forma:

```
#include <iostream>
#include <sstream>

using namespace std;
```

```
int main()
{
    istringstream is("123.456.789-10");
    string token;

    getline(is, token, '.');
    cout << token << endl;       // token = "123";

    getline(is, token, '.');
    cout << token << endl;       // token = "456";

    getline(is, token, '-');
    cout << token << endl;       // token = "789";

    getline(is, token);
    cout << token << endl;       // token = "10";

    return 0;
}
```

Para processos de tokenização mais elaborados pode ser necessário escrever um *parser* para tal. Os *parsers* serão discutidos mais adiante.

Anagramas

Anagramas são palavras formadas pelo rearranjo dos caracteres de um conjunto fixo. Por exemplo, "iracema" e "america" são anagramas, enquanto que "amora" e "roma" não são anagramas ("roma" tem os mesmo caracteres, mas não a mesma quantidade: tem um "a" a menos que "amora").

Para se determinar se determinar se duas strings `s` e `t` são anagramas há dois abordagens possíveis:

1. obter os histogramas de ambas strings e compará-los: caso sejam iguais, as strings serão anagramas;
2. ordenar ambas strings segundo a ordem lexicográfica: se após a ordenação as strings são iguais, ambas são anagramas.

Embora seja possível implementar a primeira estratégia em `O(n)`, em geral a segunda abordagem, embora seja `O(n log n)`, resulta num implementação mais simples e rápida, se o tamanho das strings for razoável (isto é, até 1 milhão de carateres).

Abaixo segue uma implementação, em C++, de uma rotina que determinar se duas strings são ou não anagramas, utilizando a segunda estratégia.

```
#include <algorithm>

bool is_anagram(const string& s, const string& t)
{
    string a(s), b(t);

    sort(a.begin(), a.end());
    sort(b.begin(), b.end());

    return a == b;
}
```

Outro problema comum é determinar o número de anagramas distintos que uma determinada palavra tem. Segundo a Análise Combinatória, este número é dado por um arranjo com repetição: se `s` tem `n` caracteres (`r` deles distintos) e `n1, n2, ..., nr` é o número de ocorrências de cada um dos `r` caracteres em `s`, então o número de anagramas distintos `A(s)` de `s` é dado por

```
A(s) = n!/(n1!n2!...nr!)
```

Para listar todos os possíveis anagramas de uma string `s`, pode-se utilizar a função `next_permutation()` da biblioteca `algorithm` do C++: ela retorna verdadeiro, e modifica a string passada, enquanto houver uma próxima permutação distinta de seus caracteres. Deve-se tomar o cuidado, porém, de ordenar a string `s` antes das sucessivas chamadas da função `next_permutation()`.

O código abaixo gera todos as 60 possíveis anagramas da palavra "banana".

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
    string s = "banana";
    int number = 0;

    sort(s.begin(), s.end());

    do {
        printf("%02d. %s\n", ++number, s.c_str());
    } while (next_permutation(s.begin(), s.end()));

    printf("%s has %d anagrams\n". s.c_str(), number);

    return 0;
}
```

Expressões Regulares

Como dito na introdução, expressões regulares (*regular expressions* ou *regex*) são uma representação que utiliza símbolos especiais para marcar sequências de caracteres ou repetições. As regex são uma forma compacta e poderosa de representar textos e padrões, mas sem o devido cuidado pode levar a *bugs* sutis relacionados a falsos positivos (o texto identificado não é o desejado) e a falsos negativos (textos desejados não são identificados).

Abaixo uma breve descrição das principais características das regexes:

1. cada caractere na regex corresponde ao mesmo caractere no texto;
2. existem combinações especiais de caracteres para corresponder à sequências de caracteres: por exemplo, `\d` corresponde a qualquer um dos dez dígitos decimais, `\D` a qualquer caractere exceto os dez dígitos decimais;
3. o caractere `.` é o coringa: ele representa qualquer caractere. O ponto final é representado pela sequência `\.` ;
4. um conjunto de caracteres válidos para o padrão pode ser representado por meio de colchetes. Por exemplo, a notação `[abc]` significa ou `a` , ou `b` . ou `c` . Se os caracteres são consecutivos, esta notação pode ser abreviada com o uso do símbolo `-` . Por exemplo, `[0-9]` tem o mesmo significado que `\d` ;
5. a notação de colchetes pode ser usada para excluir caracteres, se usada com conjunto com o símbolo `^` . Por exemplo, a notação `[^abc]` significa "todos os caracteres, exceto `a` , `b` e `c` ";
6. a sequência `\w` corresponde aos caracteres alfanuméricos `[a-zA-Z0-9_]` ;
7. Sequências especiais de caracteres podem ser usadas para representar repetições de caracteres ou padrões:
    i. um número entre chaves após o caractere/padrão indica o número de repetições ou as quantidades válidas de repetições. Por exemplo, `a{5}` significa o mesmo que `aaaaa` ; `a{1-3}` indica "de um a três caracteres `a` "; `[abc]{2}` indica dois caracteres seguidos dentres os indicados no colchete;
    ii. o caractere `*` significa "zero ou mais repetições"; o caractere `+` significa "uma ou mais repetições";
    iii. o símbolo `?` significa que o caractere ou padrão é opcional, isto é, que pode ou não ocorrer no texto. O caractere `'?'` pode ser representado pela sequência de escape `\?` ;
    iv. a sequẽncia `\s` indica espaços em branco (`' ', '\t', '\r', '\n'`);
    v. os símbolos `^` e `$` representam, respectivamente, o início e o fim do texto;
    vi. parêntesis podem ser utilizados para armazenar trechos do texto que correspondem à expressão entre parêntesis para posterior uso. Eles podem ser aninhados;
    vii. o símbolo `|` pode ser utilizado como **ou** lógico para separar grupos de padrões possíveis: a expressão `(abc|123)` significa "ou `a, b, c` ou `1, 2, 3` .

Algumas linguagens de programação, como Java e Python, tem suporte nativo para expressões regulares. A linguagem C++ incorporou suporte às regexes a partir da versão C++11, embora nem todas as implementações suportem ainda estas funcionalidades (por exemplo, ainda não há suporte no g++), e com padrão de sintaxe distinta das outras duas linguagens já citadas.

Abaixo um exemplo de um código Java que ilustra a identificação de um CPF:

```
public class Main {

    public static void main(String[] args) {
```

```
        String regex = "(\\d{3}\\.){2}\\d{3}-\\d{2}";
        String cpf = "123.456.789-10";

        System.out.println(cpf.matches(regex));      // true

        String text = "12345678910";

        System.out.println(text.matches(regex));      // false

        text = "123-456-789-10";

        System.out.println(text.matches(regex));      // false

        text = "123.456.789-1";

        System.out.println(text.matches(regex));      // false

        text = "23.456.789-10";

        System.out.println(text.matches(regex));      // false
    }
}
```

Outros métodos úteis de strings Java que podem ser utilizados com as expressões regulares são o método `replaceAll()` (que substituem os matches das regexes pelo texto dado), `trim()` (remove os espaços antes e depois do texto) e `split()` (que tokeniza a string a partir dos marcadores informados).

Exercícios

1. Codeforces
    i. 5B - Center Alignment
    ii. 32B - Borze
    iii. 41A - Translation
    iv. 58A - Chat Room
2. URI
    i. 1242 - Ácido Ribonucleico Alienígena
3. UVA
    i. 325 - Identifying Pascal Real Constants
    ii. 488 - Triangle Wave
    iii. 576 - Haiku Review
    iv. 865 - Substitution Cipher
    v. 10252 - Common Permutation

Referências

HALIM, Steve; HALIM, Felix. Competitive Programming 3, Lulu, 2013.

CPPREFERENCE. Accumulate, acesso em 24/03/2017.

CPPREFERENCE. Copy, acesso em 24/03/2017.

CPPREFERENCE. Transform, acesso em 24/03/2017.

CROCHEMORE, Maxime; RYTTER, Wojciech. Jewels of Stringology: Text Algorithms, WSPC, 2002.

REGEXONE. Lesson 1: An Introduction, and the ABCs

WIKIPEDIA. Tokenization [https://en.wikipedia.org/wiki/Tokenization_(lexical_analysis)], acesso em 22/01/17.