

Strings

Algoritmos Elementares

Prof. Edson Alves - UnB/FGA

2018

1. Palíndromos
2. Histograma
3. Strings e programação funcional
4. Tokenização
5. Anagramas
6. Expressões Regulares

Palíndromos

Definição de 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

$$P[1..N] = \text{“”} \mid P[1..1] \mid c + P[2..N - 1] + c$$

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

Identificação de palíndromos

- Uma maneira de se verificar se uma string s é ou não um palíndromo é checar se o primeiro caractere coincide com o último, o segundo com o penúltimo, e assim por diante
- Este algoritmo tem complexidade $O(|s|)$
- 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
- Isto ocorre pois se $i \geq |s|/2$, temos $i = N - 1 - j, j < |s|/2$ e a comparação de $s[i]$ com $s[N - 1 - i]$ equivale a comparação de $s[N - 1 - j]$ com $s[N - 1 - (N - 1 - j)]$, isto é, de $s[N - 1 - j]$ com $s[j], j < |s|/2$
- Mesmo que a complexidade permaneça em $O(|s|)$, esta segunda versão abaixo executa aproximadamente duas vezes mais rápido que a anterior

Implementação da rotina de identificação de palíndromos

```
1 // Observe que a função abaixo identifica s corretamente
2 // mesmo nos casos onde |s| é ímpar
3 bool is_palindrome(const string& s)
4 {
5     size_t N = s.size();
6
7     for (size_t i = 0; i < N/2; ++i)
8         if (s[i] != s[N - 1 - i])
9             return false;
10
11     return true;
12 }
```

Histograma

Definição de histograma

- Uma técnica, oriunda da estatística, que permite identificar características importantes de uma string é a construção de um histograma
- Um histograma consiste em um 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 $h['a'] = 4, h['b'] = h['c'] = h['x'] = h['i'] = 1$ e $h[y] = 0$, se $y \notin \{'a', 'b', 'c', 'x', 'i'\}$
- 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
- Os reveses são 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, onde N é o número de caracteres distintos presentes na string s)

Construção de histogramas usando o map de C++

```
1 #include <map>
2
3 std::map<char, int> histogram(const string& s)
4 {
5     std::map<char, int> h;
6
7     for (auto c : s)
8         ++h[c];
9
10    return h;
11 }
```

Construção de histogramas usando arrays estáticos

- Uma segunda forma de gerar o histograma é utilizar um array estático com 256 posições
- Estas posições cobrem todos os possíveis valores de um **char** em C/C++
- Tal abordagem assume que a string *s* contém apenas caracteres listados na tabela ASCII
- Também é preciso inicializar todas as posições deste *array* com o valor zero
- Esta construção permite atualizar/consultar os valores em $O(1)$, mas a identificação dos caracteres cujos valores associados são diferentes de zero tem que percorrer todas as 256 posições do histograma
- A abordagem anterior traria tais caracteres diretamente, sendo eles as chaves do mapa

Construção de histogramas usando arrays estáticos

```
1 #include <cstring>
2
3 void histogram(int h[256], const string& s)
4 {
5     memset(h, 0, sizeof h);
6
7     for (auto c : s)
8         ++h[c];
9 }
```

Construção de histogramas com mapeamento do alfabeto

- A terceira e última abordagem 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, por exemplo, o alfabeto for constituído das letras maiúsculas e minúsculas, esta indexação é feita de forma direta, em $O(1)$
- Caso contrário, é preciso procurar pelo caractere no alfabeto em $O(M)$ (ou $O(\log M)$, 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

Construção de histogramas com mapeamento do alfabeto

```
1 // Esta implementação assume que o alfabeto é composto
2 // pelas 26 letras maiúsculas do alfabeto
3 void histogram(int h[26], const string& s)
4 {
5     memset(h, 0, sizeof h);
6
7     for (auto c : s)
8         ++h[c - 'A'];
9 }
```

Strings e programação funcional

- 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 caracteres
- A vantagem de tal abordagem é a redução do tamanho do código, evitando laços e variáveis temporárias explícitas
- Outro aspecto importante é que o uso de tais conceitos permitem simplificar a notação e descrição de problemas de strings

- Um mapa (ou mapeamento) consiste em uma função $m_f : S_N \rightarrow S_N$, onde S_N é o conjunto de todas as strings de tamanho N e $f : A \rightarrow A$ 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_f 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
- A implementação do mapeamento pode ser apenas conceitual, usando uma função padrão do C/C++, ou utilizar a função `transform()` da STL da linguagem C++

Exemplo de implementação de um mapa usando funções

```
1 #include <functional>
2 #include <iostream>
3 #include <cctype>
4
5 using namespace std;
6
7 // O nome smap ('string map') foi utilizado para evitar
8 // confusão com a classe map da STL
9 string smap(const string& s, const function<char(char)>& f)
10 {
11     string y;
12
13     for (const auto& c : s)
14         y.push_back(f(c));
15
16     return y;
17 }
18
```

Exemplo de implementação de um mapa usando funções

```
19 int main()
20 {
21     string s = "Teste de mapeamento";
22     auto y = smap(s, [](char c) { return (char) toupper(c); });
23
24     cout << y << '\n';
25
26     return 0;
27 }
```

Implementação da cifra de César usando a função transform()

```
1 #include <iostream>
2 #include <algorithm>
3
4 using namespace std;
5
6 int main()
7 {
8     string text { "cesar cipher xyz" };
9     string cipher;
10
11     transform(text.begin(), text.end(), back_inserter(cipher),
12               [](char c) {
13                 return c == ' ' ? c : (((c - 'a') + 3) % 26) + 'a';
14             });
15
16     cout << cipher << endl;
17
18     return 0;
19 }
20 }
```

- Um filtro $f_p : S_N \rightarrow S_M$ consiste em uma função que gera uma string de tamanho $M \leq N$, a partir de uma string de tamanho N , através da seleção dos M caracteres cujo predicado $p : A \rightarrow Bool$ retorna verdadeiro
- Assim como os mapas, os filtros podem ser implementados de forma apenas conceitual, usando elementos da própria linguagem, como laços e condicionais
- Outra maneira de implementar o mesmo código é utilizar a função `copy_if()` da STL, que tem sintaxe semelhante a da função `transform()`
- A função `remove_copy_if()` tem comportamento análogo, mas copia os caracteres que negarem o predicado

Implementação de um filtro usando elementos de C++

```
1 #include <cctype>
2
3 bool is_vowel(char c)
4 {
5     const string vowels { "aeiou" };
6
7     return vowels.find(tolower(c)) != string::npos;
8 }
9
10 // Extrai apenas as vogais de s
11 string filter_vowels(const string& s)
12 {
13     string v;
14
15     for (auto c : s)
16         if (is_vowel(c))
17             v.push_back(c);
18
19     return v;
20 }
```

Implementação de um filtro usando a função copy_if()

```
1 #include <cctype>
2 #include <algorithm>
3
4 // Extrai apenas as vogais de s
5 string filter_vowels(const string& s)
6 {
7     string v;
8
9     copy_if(s.begin(), s.end(), back_inserter(v),
10            [](char c)
11            {
12                const string vowels { "aeiou" };
13
14                return vowels.find(c) != string::npos;
15            }
16    );
17
18    return v;
19 }
```

Reduções

- Uma redução $r_b : S_N \rightarrow T$ 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 operando esquerdo inicial um valor definido previamente
- Se $b(s_i, s_j) = t_k$ e s_0 é o valor inicial para o operando esquerdo, então a redução se comporta da seguinte maneira na sequência $s = \{s_1, s_2, \dots, s_N\}$:

$$\begin{aligned} r_b(s) &= \{b(s_0, s_1), s_2, \dots, s_N\} \\ &= \{b(t_1, s_2), \dots, s_N\} \\ &= \{b(t_2, s_3), \dots, s_N\} \\ &= \dots \\ &= \{b(t_{N-1}, s_N)\} = t_N \end{aligned}$$

Reduções

- Reduções podem ser implementadas de maneira natural usando um laço e um valor acumulador
- Uma alternativa é utilizar a função `accumulate()` da STL, que abstrai o conceito de redução, parametrizando o tipo de retorno e o valor inicial do operando esquerdo
- No padrão C++17 foi introduzida a função `reduce()`
- O comportamento é semelhante ao da função `accumulate()`, mas ela não impõem uma ordem na aplicação do operador binário
- Este relaxamento permite otimizações por parte do compilador e a execução em paralelo
- Contudo, nem todos os compiladores implementaram a função `reduce()` ainda

Implementação de uma redução usando laço e acumulador

```
1 // Soma todos os dígitos da strings de dígitos s
2 int sum(const string& s)
3 {
4     int s = 0;
5
6     for (auto c : s)
7         s += (c - '0');
8
9     return s;
10 }
```

Implementação de uma redução usando a função accumulate()

```
1 #include <iostream>
2 #include <numeric>
3
4 using namespace std;
5
6 // Soma todos os dígitos da strings de dígitos s
7 // O operador binário default é a soma
8 int sum(const string& s)
9 {
10     return accumulate(s.begin(), s.end(), 0,
11         [](int a, char b) { return a + (b - '0'); });
12 }
13
14 int main()
15 {
16     cout << sum("12345") << '\n';
17
18     return 0;
19 }
```

Tabelas de substituição

- Uma tabela de substituição é uma aplicação prática de um mapeamento
- Ela é definida por uma função $f : A \rightarrow A$ 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 e a criptografia baseada em ou exclusivo (*xor*)
- Se $|A|$ não é muito grande, a função f pode ser implementada em um *array* estático, permitindo a consulta da substituição com complexidade $O(1)$
- Caso contrário, ou se o alfabeto não é contíguo, deve ser usado um dicionário para armazenar tais substituições

Exemplo de uso de tabela de substituição

```
1 // Exemplo de criptografia baseada em xor
2 #include <iostream>
3 #include <algorithm>
4
5 using namespace std;
6
7 const int MAX { 256 };
8 char table[MAX];
9
10 string cipher(const string& text)
11 {
12     string res;
13
14     transform(text.begin(), text.end(), back_inserter(res), [](char c)
15     {
16         return table[(int) c];
17     }
18 );
19
20     return res;
21 }
```

Exemplo de uso de tabela de substituição

```
23 void fill_table(char key)
24 {
25     for (int i = 0; i < MAX; ++i)
26         table[i] = i ^ key;
27 }
28
29 string decipher(const string& c)
30 {
31     return cipher(c);
32 }
33
34 int main()
35 {
36     fill_table(0x3B);
37
38     string message { "Xor cipher example" };
39     string c = cipher(message);
40
41     printf("c = ");
42     for (size_t i = 0; i < c.size(); ++i)
43         printf("%02x%c", c[i], (i + 1 == c.size() ? '\n' : ' '));
```

Exemplo de uso de tabela de substituição

```
44
45     string d = decipher(c);
46
47     printf("d = [%s]\n", d.c_str());
48
49     return 0;
50 }
```

Tokenização

Definição

- 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 de um delimitador
- Por exemplo, a frase “O rato roeu a roupa do rei de Roma” seria dividida em 9 tokens: “O”, “rato”, “roeu”, “a”, “roupa”, “do”, “rei”, “de”, “Roma”

strtok()

- 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 exista mais tokens
- Esta função tem um comportamento incomum, devido três aspectos importantes:
 1. ela mantém, internamente, um ponteiro para o início do próximo *token*
 2. esta função altera o parâmetro de entrada, escrevendo o caractere `'\0'` nas posições onde são encontrados delimitadores
 3. a função `strtok()` não retorna *tokens* vazios (isto é, de tamanho zero).

strtok()

- Devido ao aspecto 1, 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*
- Já em relação ao aspecto 2, se for necessário preservar o conteúdo original da string, deve ser passada uma cópia da mesma como primeiro parâmetro
- Por fim, por conta do aspecto 3, a chamada de strtok() na string "aaaaa" com delimitador "a" retorna **NULL** já na primeira chamada
- A biblioteca string.h oferece uma versão *thread safe* de strtok(), denominada strtok_r()
- Nesta função há um terceiro parâmetro, que é utilizado para memorizar a posição do próximo *token*

Exemplo de uso da função strtok()

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main() {
5     char cpf[] = "123.456.789-10", *token = NULL;
6
7     token = strtok(cpf, ".");
8     printf("token = [%s]\n", token);    /* token = "123" */
9
10    token = strtok(NULL, ".");
11    printf("token = [%s]\n", token);    /* token = "456" */
12
13    token = strtok(NULL, "-");
14    printf("token = [%s]\n", token);    /* token = "798" */
15
16    token = strtok(NULL, "\\0");
17    printf("token = [%s]\n", token);    /* token = "10" */
18    printf("cpf = %s\n", cpf);          /* cpf = "123" */
19
20    return 0;
21 }
```

Tokenização em C++

- 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)
- O segundo parâmetro é a string que conterá o *token* identificado
- O terceiro parâmetro é o caractere delimitador
- Se omitido, o caractere `'\n'` será considerado o delimitador
- Esta função é útil quando há apenas um delimitador
- Para processos de tokenização mais elaborados pode ser necessário escrever um *parser*

Exemplo de uso da função getline()

```
1 #include <iostream>
2 #include <sstream>
3
4 int main() {
5     std::istringstream is("123.456.789-10");
6     std::string token;
7
8     getline(is, token, '.');
9     std::cout << token << '\n';      // token = "123";
10
11     getline(is, token, '.');
12     cout << token << '\n';          // token = "456";
13
14     getline(is, token, '-');
15     cout << token << '\n';          // token = "789";
16
17     getline(is, token);
18     cout << token << '\n';          // token = "10";
19
20     return 0;
21 }
```

Anagramas

Definição e identificação de 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 duas 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

Implementação da verificação de anagramas

```
1 #include <algorithm>
2
3 bool is_anagram(const string& s, const string& t)
4 {
5     string a(s), b(t);
6
7     sort(a.begin(), a.end());
8     sort(b.begin(), b.end());
9
10    return a == b;
11 }
```


Listagem de todos os anagramas

- Um 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 n_1, n_2, \dots, n_r 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

$$A(s) = \frac{n!}{n_1!n_2!\dots n_r!}$$

- 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()`

Exemplo de listagem de todos os anagramas de uma palavra

```
1 #include <iostream>
2 #include <algorithm>
3
4 using namespace std;
5
6 int main()
7 {
8     string s { "banana" };
9     int number = 0;
10
11     sort(s.begin(), s.end());
12
13     do {
14         printf("%02d. %s\n", ++number, s.c_str());
15     } while (next_permutation(s.begin(), s.end()));
16
17     printf("%s has %d anagrams\n", s.c_str(), number);
18
19     return 0;
20 }
```

Expressões Regulares

- Expressão regular (*regular expression* ou *regex*) é uma representação que utiliza símbolos especiais para marcar sequências de caracteres ou repetições
- As *regexes* são uma forma compacta e poderosa de representar textos e padrões
- Porém, 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)
- 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, com padrão de sintaxe distinta das outras duas linguagens já citadas

Características de uma regex

1. cada caractere da *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`

Características de uma regex

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:
 - 7.1 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 dentre os indicados no colchete
 - 7.2 o caractere `*` significa “zero ou mais repetições”
 - 7.3 o caractere `+` significa “uma ou mais repetições”

Características de uma regex

- 7.4 o símbolo `?` significa que o caractere ou padrão é opcional, isto é, que pode ou não ocorrer no texto
- 7.5 O caractere `'?'` pode ser representado pela sequência de escape `\?`
- 7.6 a sequência `\s` indica espaços em branco (`' '`, `'\t'`, `'\r'`, `'\n'`)
- 7.7 os símbolos `^` e `$` representam, respectivamente, o início e o fim do texto
- 7.8 parêntesis podem ser utilizados para armazenar trechos do texto que correspondem à expressão entre parêntesis para posterior uso. Eles podem ser aninhados
- 7.9 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"

Exemplo de uso de regex em Java

```
1 public class Main {  
2  
3     public static void main(String[] args) {  
4         String regex = "(\\d{3}\\\\.){2}\\d{3}-\\d{2}";  
5         String cpf = "123.456.789-10";  
6  
7         System.out.println(cpf.matches(regex));        // true  
8  
9         String text = "12345678910";  
10        System.out.println(text.matches(regex));        // false  
11  
12        text = "123-456-789-10";  
13        System.out.println(text.matches(regex));        // false  
14  
15        text = "123.456.789-1";  
16        System.out.println(text.matches(regex));        // false  
17  
18        text = "23.456.789-10";  
19        System.out.println(text.matches(regex));        // false  
20    }  
21 }
```


1. CppReference. [Accumulate](#), acesso em 24/03/2017.
2. CppReference. [Copy](#), acesso em 24/03/2017.
3. CppReference. [Transform](#), acesso em 24/03/2017.
4. CppReference. [Reduce](#), acesso em 14/03/2019.
5. **CROCHEMORE**, Maxime; **RYTTER**, Wojciech. *Jewels of Stringology: Text Algorithms*, WSPC, 2002.
6. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.
7. RegexOne. [Lesson 1: An Introduction, and the ABCs](#), acesso em 15/03/2019.
8. Wikipedia. [Tokenização](#), acesso em 22/01/2017.