Strings

Algoritmos Elementares

Prof. Edson Alves - UnB/FGA

2018

Sumário

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 ínicio 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 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 $h[\text{`}a\text{'}] = 4, h[\text{`}b\text{'}] = h[\text{`}c\text{'}] = h[\text{`}x\text{'}] = h[\text{`}i\text{'}] = 1$ e $h[y] = 0$, se $y \notin \{\text{`}a\text{'}, \text{`}b\text{'}, \text{`}c\text{'}, \text{`}x\text{'}, \text{`}i\text{'}\}$
- 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 revezes 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++

```cpp
#include <map>

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

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

    return h;
}
```

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

```
#include <cstring>

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

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

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

Motivação

- 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

Mapas

- Um mapa (ou mapeamento) consiste em uma função $m_f : S_N \to S_N$, onde S_N é o conjunto de todas as strings de tamanho N e $f : A \to A$ 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

```cpp
#include <functional>
#include <iostream>
#include <cctype>

using namespace std;

// O nome smap ('string map') foi utilizado para evitar
// confusão com a classe map da STL
string smap(const string& s, const function<char(char)>& f)
{
    string y;

    for (const auto& c : s)
        y.push_back(f(c));

    return y;
}

```

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()

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

using namespace std;

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

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

    cout << cipher << endl;

    return 0;
}
```

Filtros

- Um filtro $f_p : S_N \to S_M$ 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 $p : A \to Bool$ 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++

```cpp
#include <cctype>

bool is_vowel(char c)
{
    const string vowels { "aeiou" };

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

// Extrai apenas as vogais de s
string filter_vowels(const string& s)
{
    string v;

    for (auto c : s)
        if (is_vowel(c))
            v.push_back(c);

    return v;
}
```

Implementação de um filtro usando a função copy_if()

```cpp
#include <cctype>
#include <algorithm>

// Extrai apenas as vogais de s
string filter_vowels(const string& s)
{
    string v;

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

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

    return v;
}
```

Reduções

- Uma redução $r_b : S_N \to T$ 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 $s = \{s_1, s_2, \ldots, s_N\}$:

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

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()

```cpp
#include <iostream>
#include <numeric>

using namespace std;

// Soma todos os dígitos da strings de dígitos s
// O operador binário default é a soma
int sum(const string& s)
{
    return accumulate(s.begin(), s.end(), 0,
        [](int a, char b) { return a + (b - '0'); });
}

int main()
{
    cout << sum("12345") << '\n';

    return 0;
}
```

Tabelas de substituição

- Uma tabela de substituição é uma aplicação prática de um mapeamento
- Ela é definida por uma função $f : A \to A$ 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

```cpp
// Exemplo de criptografia baseada em xor
#include <iostream>
#include <algorithm>

using namespace std;

const int MAX { 256 };
char table[MAX];

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

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

    return res;
}
```

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  }
```

Referências

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.